UNIVERSIDADE D COIMBRA

João Rodrigues de Campos

Advanced Online Failure Prediction Through Machine Learning

Tese no âmbito do Programa de Doutoramento em Ciências e Tecnologias da Informação, orientada pelo Professor Doutor Marco Paulo Amorim Vieira e pelo Professor Doutor Ernesto Jorge Fernandes Costa, e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Novembro de 2021

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Advanced Online Failure Prediction Through Machine Learning

João Rodrigues de Campos

Doctoral Program in Information Science and Technology
PhD Thesis submitted to the University of Coimbra

Advised by Professor Marco Paulo Amorim Vieira
and Professor Ernesto Jorge Fernandes Costa

November, 2021

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

Previsão de Avarias Através de Aprendizagem Computacional

João Rodrigues de Campos

Programa de Doutoramento em Ciências e Tecnologias da Informação
Tese de Doutoramento apresentada à Universidade de Coimbra

Orientado pelo Professor Doutor Marco Paulo Amorim Vieira
e pelo Professor Doutor Ernesto Jorge Fernandes Costa

Novembro, 2021

This work was partially supported by the by the Portuguese Foundation for Science and Technology (FCT) under PhD grant SFRH/BD/140221/2018 and project grants nº 777154, POCI-01-0145-FEDER-032504, and POCI-01-0247-FEDER-045907.

Everything is obvious,
once you know the answer

- Duncan J. Watts

Acknowledgments

Successfully finishing a Ph.D. is only possible with the collective support of family, friends, and supervisors (which I am also glad to consider among my friends). A special thanks to all of you. Besides the non-negligible effort required to execute a meaningful Ph.D., these last few years were also very demanding at a personal level. Balancing professional goals, being a good and present father and husband, keeping family and friends close, and preserving my sanity throughout the process was a constant challenge. Notwithstanding, I knew I could always count on all of you.

I want to especially thank my advisors and my friends, Professor Marco Vieira and Professor Ernesto Costa. Thank you for inviting and accepting the challenge of mentoring me through this thesis. Your valuable (mis)guidance and knowledge were invaluable to solve the various problems that we came across. While, as a good pupil, I might not have always done what you would have liked, rest assured that I always took your advice into consideration. These years allowed me to get to know both of you more personally. Your knowledge, experience, and availability to often discuss subjects other than the thesis enabled me to also grow as a person.

I would also, and above all, like to thank my wife for her support, patience, and maintaining a *home* while I was wandering in my research. I know it has not been easy (and it will not likely be soon either) but this would not have been possible without you. A very special and warm thanks to my daughter Matilde, whose laughs, hugs, and unconditional love gave me the strength to withstand the long days and nights of strenuous work. A special appreciation to my parents as well, who have always instilled in me the importance and privilege of *knowledge*. Thanks to Bruno Pais and Manuel Vaz Pinto for providing a safe haven of fellowship to escape the never-ending problems of research. Know that I consider you as family.

Finally, the context in which this thesis was conducted, the COVID-19 pandemic, meant that it was not possible to foster new connections and discussions at conferences or international exchanges. Unfortunately, this also affected the invaluable discussions and relations that would have otherwise taken place at the university. Notwithstanding, with the risk of forgetting someone, thank you to everyone that somehow was a part of my Ph.D.. A special thanks to the members of the Software and Systems Engineering (SSE) group, especially Gonçalo Carvalho, José Pereira, Charles Gonçalves, and Frederico Cerveira, who have been with me since the beginning of this journey. A thank you note also to Luís Ventura, Inês Valentim, Ana Duarte, Kevin Bento, and Noé Godinho. Although COVID-19 did not allow us to meet as often in these last years, I always knew I could talk to you.

Abstract

The complexity of modern software makes it nearly impossible to detect every fault before deployment. Such faults can ultimately lead to failures at runtime, compromising the business process and leading to non-negligible costs or losses. Online Failure Prediction (OFP) is a fault-tolerance technique that attempts to predict incoming failures to avoid or mitigate their consequences based on the analysis of past data and the current system state. The systems that can benefit the most from these advances are large servers, including both cloud and virtualized systems, where the costs and risks associated with a failure are not negligible. At the same time, given recent technological developments, Machine Learning (ML) algorithms have shown their ability to adapt and extract knowledge in a variety of complex problems. Although there are already some works relying on ML for OFP, most of them focus on local experiments with a small set of ML techniques or targeting smaller components (e.g. hard-drives).

Given the reliability of modern systems, one challenge to the widespread use of OFP is that failures are rare events, and thus adequate failure data for training predictors is typically not available. To overcome this issue, fault injection has been accepted as a viable alternative to generate realistic failure data. However, despite the vast literature, fault injectors are still complex pieces of software that are difficult to implement and use. Generating data through fault injection also raises some challenges, such as how to properly process the data to create and assess predictive models. The particular characteristics of fault injection campaigns combined with those of OFP require specific considerations and techniques to create representative predictors.

Another reason why OFP is not widely adopted is that processes and tools for assessing and comparing failure prediction solutions are not available. Effectively implementing failure prediction involves extremely accurate tuning, but also an adequate selection of the most suitable models for a specific target system. More precisely, selecting a particular predictor requires a rigorous assessment of alternative solutions using appropriate metrics, and their comparison using well-defined processes. Considering the complexity and interdisciplinarity of the various techniques and research fields required to create accurate failure predictors, the existing body of related work is very limited. As a result, research and development on OFP for complex systems have become stale over the years.

This thesis addresses the most relevant challenges to the use of OFP, by thoroughly exploring the problem and proposing a comprehensive framework to support the development of predictive models. The key contributions can be divided into two groups. The first comprises *techniques and artifacts to support OFP*, including a detailed procedure on the use of fault injection to generate failure data to support OFP on modern systems. To make such a process feasible, guidelines are provided on how to implement a testbed for complex experimental processes that can leverage modern computational power while ensuring the consistency and repeatability of experiments. Also, given the plethora and complexity of ML methods and the constant need to explore new problems and techniques, a customizable and comprehensive ML toolbox was implemented to support the exploration and development of predictive models.

The second group of contributions focuses on *procedures and methodologies to develop OFP solutions*. This includes a thorough methodology on how to use failure data generated using fault injection to properly develop predictive models, taking into account the specific characteristics of the failure prediction domain. As properly assessing and comparing the performance of ML models requires a rigorous and well-defined process that takes into consideration the operational needs of the system, a benchmarking approach for ML-based solutions for OFP is proposed. The ML techniques considered, used, and implemented throughout the work are in themselves significant contributions due to the extent and detail into which they were analyzed and studied for the OFP problem. Ultimately, this yields innovation on how to use and assess advanced ML solutions for problems with similar characteristics.

To demonstrate usefulness and effectiveness, the framework was used to explore and create accurate failure predictors for a modern system. This comprised an extensive fault injection campaign on a system based on an up-to-date Linux kernel, considering different workloads representing different usage scenarios, several fault types, and various failure modes. These data were then thoroughly explored and an extensive study was conducted on the use of ML techniques, from traditional to state-of-the-art algorithms, and their applicability to OFP. Also, a comprehensive benchmarking campaign was conducted to achieve a fair and sound comparison of alternative solutions, exploring the need to consider the technical requirements of the system where the predictors will operate. Results demonstrate that not only it is possible to create accurate failure predictors, but also that the techniques, artifacts, methodologies, and procedures proposed are essential to guide the process and assure a representative and sound experimental process.

Keywords: Online Failure Prediction, Fault Injection, Dependability, Machine Learning, Benchmarking

Resumo

A crescente complexidade de produtos de *software* torna praticamente impossível detetar todas as falhas antes de estes serem postos em produção. Estas falhas podem levar a avarias durante a execução, o que compromete todo o processo de negócio e acarreta custos ou perdas elevadas. Previsão de Avarias (PA) é uma técnica de tolerância a falhas que tenta prever a ocorrência de avarias num futuro próximo de modo a evitar ou mitigar as suas consequências, com base na análise de dados de avarias ocorridas no passado e no estado atual do sistema. Os sistemas que mais podem beneficiar com avanços nesta área são os servidores, incluindo sistemas em nuvem e virtualizados, em que os custos e riscos associados a uma avaria não são negligenciáveis. Ao mesmo tempo, devido aos recentes desenvolvimentos tecnológicos, os algoritmos de Aprendizagem Computacional (AC) têm demonstrado capacidade de extrair conhecimento em múltiplos problemas complexos. Embora já existam alguns trabalhos que aplicam AC para PA, a maioria restringe-se a experiências *ad-hoc*, com um conjunto muito limitado de técnicas de AC, ou é focado em componentes de pequena escala (e.g., discos rígidos).

Dada a confiabilidade dos sistemas atuais, as avarias são eventos raros, o que constitui um desafio para o uso de PA, uma vez que dificilmente existem dados suficientes para desenvolver modelos de previsão. Para ultrapassar esse problema, a injeção de falhas foi aceite como uma alternativa viável para gerar dados de avarias realistas. No entanto, apesar da vasta literatura existente, os injetores de falhas ainda são ferramentas complexas, difíceis de implementar e usar. De facto, a geração de dados através de injeção de falhas levanta alguns desafios, como, por exemplo, processar os dados adequadamente para criar e avaliar modelos preditivos. As características de dados gerados através de injeção de falhas combinadas com as de PA requerem considerações e técnicas específicas para que os modelos criados sejam representativos.

Outra razão pela qual PA não é amplamente adotada é a inexistência de processos e ferramentas adequados para avaliar e comparar as soluções de previsão de avarias. A implementação eficaz da previsão de avarias requer um ajuste extremamente preciso, mas também uma seleção sistematizada dos modelos mais apropriados para um dado sistema. Mais concretamente, a seleção de um modelo exige uma avaliação rigorosa de soluções alternativas com recurso a métricas adequadas e à sua comparação através de processos bem definidos. Considerando a complexidade e interdisciplinaridade das várias técnicas e campos de pesquisa necessários para criar preditores de avarias, o número de trabalhos relacionados é muito limitado. Como resultado, a investigação e o desenvolvimento em PA para sistemas complexos estagnaram com o passar dos anos.

Esta tese aborda os desafios mais relevantes para o uso de PA, explorando pormenorizadamente o problema e propondo uma abordagem para apoiar o desenvolvimento de modelos preditivos. As contribuições principais podem ser divididas em dois grupos. O primeiro compreende técnicas e ferramentas de suporte ao desenvolvimento para PA, incluindo um procedimento pormenorizado sobre o uso de injeção de falhas para gerar dados de avarias. Para tornar esse processo viável, são fornecidas diretrizes sobre como implementar um ambiente de teste que permita alavancar o poder computacional atual, garantindo a consistência e repetibilidade das experiências. Além disso, dada a abundância e complexidade dos métodos de AC e a necessidade constante de explorar novos problemas e técnicas, é apresentada uma ferramenta extensível de AC para apoiar a exploração e o desenvolvimento de modelos preditivos.

O segundo grupo de contribuições, por sua vez, centra-se em procedimentos e metodologias para desenvolver soluções de PA, incluindo uma metodologia para a utilização de dados de avarias gerados através de injeção de falhas no desenvolvimento de modelos preditivos, levando em consideração as características específicas do domínio de previsão de avarias. Uma vez que avaliar e comparar adequadamente o desempenho dos modelos de AC requer um processo rigoroso e bem definido que leve em consideração as necessidades operacionais do sistema, é proposta uma abordagem de *benchmarking* para soluções baseadas em AC para PA. Para além disso, as técnicas de AC consideradas, utilizadas e implementadas ao longo do trabalho são, em si mesmas, contribuições significativas devido à extensão e pormenores em que foram analisadas e estudadas para o problema de OPF. Isto gera inovação sobre como usar e avaliar soluções avançadas de AC para problemas com características semelhantes.

Para demonstrar a utilidade e eficácia, a abordagem proposta foi usada para explorar e criar preditores de avarias para um sistema moderno. Tal inclui uma ampla campanha de injeção de falhas num sistema baseado num kernel Linux atualizado, considerando diferentes cargas de trabalho que representam diferentes cenários de utilização, vários tipos de falhas e diversos modos de avarias. Esses dados foram posteriormente explorados e foi realizado um estudo aprofundado sobre o uso de técnicas de AC, desde algoritmos tradicionais até algoritmos de última geração, e a sua aplicabilidade para PA. Além disso, foi conduzida uma campanha extensa de *benchmarking* para comparar de forma justa as soluções alternativas, explorando a necessidade de considerar os requisitos técnicos do sistema no qual os preditores irão funcionar. Os resultados demonstram que não só é possível criar preditores de avarias precisos, mas também que as técnicas, ferramentas, metodologias e procedimentos propostos nesta tese são essenciais para guiar o processo e garantir um processo experimental representativo.

Palavras-chave: Previsão de Avarias, Injeção de Falhas, Confiabilidade, Aprendizagem Computacional, Benchmarking

Foreword

The work detailed in this thesis was accomplished at the Software and Systems Engineering (SSE) group of the Centre for Informatics and Systems of the University of Coimbra (CISUC), within the context of the following projects and grants:

- **ATMOSPHERE** – Adaptive, Trustworthy, Manageable, Orchestrated, Secure Privacy-assuring Hybrid, Ecosystem for REsilient Cloud Computing; a 24-month project aiming at the design and development of an ecosystem of a framework, platform and application of next generation trustworthy cloud services on top of an intercontinental hybrid and federated resource pool. The framework considers a broad spectrum of properties and their measures. The platform supports the building, deployment, measuring and evolution of trustworthy cloud resources, data network and data services. ATMOSPHERE is funded by the European Union under the Cooperation Programme, Horizon 2020 grant agreement No 777154.
- **METRICS** – Monitoring and Measuring the Trustworthiness of Critical Cloud Systems; cloud is pervasive nowadays, but its adoption in critical systems is limited by trust issues, mainly influenced by security, dependability, privacy, fairness and transparency concerns, as per the recent GDPR regulation. As an evolving concept, the trustworthiness of the system must be continuously monitored and measured, but there is a lack of means to do that in cloud environment. This project aims to propose a framework and means for monitoring and assessing the trustworthiness of cloud systems. This includes the definition of trustworthiness properties, their continuous measurement and analysis. METRICS is co-funded by the Portuguese Foundation for Science and Technology (FCT) and by the Fundo Europeu de Desenvolvimento Regional (FEDER) through Portugal 2020 - Programa Operacional Competitividade e Internacionalização (POCI-01-0145-FEDER-032504).
- **AIDA** – Adaptive, Intelligent and Distributed Assurance Platform; the project aims at improving a platform used by Mobileum for integral risk management in companies. This platform ensures revenue, corporate conditions and fraud control for companies. Thanks to the newest version of the platform, developed by AIDA, companies will be able to collect and monitor data in an extremely flexible way, with real-time guarantees, security and reliability. AIDA is co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalization – COMPETE 2020 (POCI-01-0247-FEDER-045907) and by the Portuguese Foundation for Science and Technology under CMU Portugal Program.
- **Ph.D. grant** – Foundation for Science and Technology (FCT) grant SFRH/BD/140221/2018.

- **Research grant** – Centre for Informatics and Systems of the University of Coimbra (CISUC) grant DPA 18-034.

The contributions of this thesis resulted in several publications in international peer-reviewed conferences (6) and journals (1):

1. J. R. Campos, M. Vieira, and E. Costa, 'Exploratory Study of Machine Learning Techniques for Supporting Failure Prediction'. In: 2018 14th European Dependable Computing Conference (EDCC). IEEE, 2018. p. 9-16 (*Best paper award*).
2. J. R. Campos, M. Vieira, and E. Costa, 'Propheticus: Machine learning Framework for the Development of Predictive Models for Reliable and Secure Software'. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2019. p. 173-182.
3. J. R. Campos, E. Costa, and M. Vieira, 'Improving Failure Prediction by Ensembling the Decisions of Machine Learning Models: A Case Study'. IEEE Access, 2019, 7: 177661-177674.
4. J. D. Pereira, J. R. Campos, and M. Vieira, 'An Exploratory Study on Machine Learning to Combine Security Vulnerability Alerts from Static Analysis Tools'. In: 2019 9th Latin-American Symposium on Dependable Computing (LADC). IEEE, 2019. p. 1-10.
5. J. R. Campos, E. Costa, and M. Vieira, 'On Configuring a Testbed for Dependability Experiments: Guidelines and Fault Injection Case Study'. In: International Conference on Computer Safety, Reliability, and Security. Springer, Cham, 2020. p. 419-433.
6. J. R. Campos, and Ernesto Costa. 'Fault Injection to Generate Failure Data for Failure Prediction: A Case Study'. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2020. p. 115-126.
7. J. D. Pereira, J. R. Campos, and M. Vieira, 'Machine Learning to Combine Static Analysis Alerts with Software Metrics to Detect Security Vulnerabilities: An Empirical Study'. In: 2021 17th European Dependable Computing Conference (EDCC). IEEE, 2021, (*Best paper award*) ***in press***

Two additional works have been submitted to journals and are currently under review:

1. J. R. Campos, E. Costa, and M. Vieira, 'A Methodology for Machine Learning-based Online Failure Prediction', submitted in *Transactions on Computers*
2. J. R. Campos, E. Costa, and M. Vieira, 'A Benchmarking Approach for Machine Learning-based Online Failure Prediction', submitted in *Transactions for Dependable and Secure Computing*

Contents

Acknowledgments xi

Abstract xiii

Resumo xv

Foreword xvii

List of Figures xxiv

List of Tables xxv

Acronyms xxvii

1 Introduction **1**
- 1.1 Problem Statement . . . 2
- 1.2 Drivers and Framework . . . 5
- 1.3 Contributions . . . 6
- 1.4 Outline of the Document . . . 8

2 Background and Related Work **11**
- 2.1 Dependability . . . 11
  - 2.1.1 Online Failure Prediction (OFP) . . . 13
  - 2.1.2 Fault Injection . . . 15
  - 2.1.3 Dependability and Experimental Testbeds . . . 19
  - 2.1.4 Benchmarking . . . 20
- 2.2 Machine Learning (ML) . . . 23
  - 2.2.1 Data Analysis . . . 24
  - 2.2.2 Data Preparation . . . 25
  - 2.2.3 Learning Types and Algorithms . . . 26
  - 2.2.4 Evaluation . . . 30
  - 2.2.5 Time Series . . . 33
  - 2.2.6 Adversarial Machine Learning (ML) . . . 35
- 2.3 Related Work . . . 37
  - 2.3.1 Fault Injection and Dependability Testbeds . . . 37
  - 2.3.2 Benchmarking . . . 39
  - 2.3.3 Machine Learning (ML) and Dependability . . . 40
  - 2.3.4 Online Failure Prediction (OFP) . . . 42
- 2.4 Summary . . . 45

3 Techniques and Artifacts to Support OFP 47
3.1 Configuring and Deploying a Testbed 48
3.1.1 Drivers 49
3.1.2 Simultaneous Execution and Isolation 51
3.1.3 Using Virtualization 54
3.1.4 Monitorization 54
3.1.5 Process Automation 55
3.2 Fault Injection to Generate Failure Data 55
3.2.1 Preparation 57
3.2.2 Generate Data 58
3.2.3 Process Data 60
3.3 Propheticus: Machine Learning Toolbox 60
3.3.1 Overall Architecture 62
3.3.2 Implementation 63
3.3.3 Functionalities/Modules 63
3.3.4 Instantiation and Configuration 64
3.4 Summary 65

4 Methodology for Developing Predictive Models 67
4.1 Generate Failure Data 68
4.2 Process, Cleanse, Augment Data 69
4.3 Parse Failures and Define Failure Classes 71
4.4 Tune Models per Failure/Class 72
4.5 Analyze Experiment-wise Performance 76
4.6 Deploy the Best Model per Failure Class 77
4.7 Summary 78

5 A Benchmarking Approach for ML-based OFP 79
5.1 Scenarios 81
5.2 Metrics 82
5.3 Algorithms 84
5.4 Dataset/Workload 85
5.5 Procedure 85
5.5.1 Preparation 86
5.5.2 Dataset building and validation 86
5.5.3 Execution of the prediction algorithms 87
5.5.4 Metrics calculation, assessment, and comparison 87
5.5.5 Robustness assessment 88
5.6 Summary 89

6 Experimental Evaluation 91
6.1 Generating Failure Data 91
6.1.1 Experimental Setup 91
6.1.2 Workloads 92
6.1.3 Assessing Isolation 93
6.1.4 Fault Injector and Fault Models 96
6.1.5 Failure Modes 97
6.1.6 Discussion 98

6.2 Creating Failure Predictors 102
6.2.1 Data Preparation 102
6.2.2 ML Algorithms and Techniques 104
6.2.3 Binary Classification 107
6.2.4 Classification by Failure Mode 108
6.2.5 Discussion 112
6.3 Developing Predictive Models 114
6.3.1 Generate Failure Data 115
6.3.2 Process, Cleanse, Augment Data 115
6.3.3 Parse Failures and Define Failure Classes 116
6.3.4 Tune Models per Failure/Class 117
6.3.5 Analyze Experiment-wise Performance 121
6.3.6 Deploy Best Model per Failure Class 122
6.3.7 Discussion 122
6.4 Benchmarking Failure Predictors 124
6.4.1 Preparation 125
6.4.2 Dataset Building and Validation 126
6.4.3 Execution of Prediction Algorithms and Metrics Calculation, Assessment, and Comparison 127
6.4.4 Assessing Robustness 130
6.4.5 Discussion 133
6.5 Summary 134

7 Conclusions and Future Work **137**

Bibliography **141**

Appendixes **161**
Appendix A: OFP on Windows OS 161
A.1 Dataset 161
A.2 ML Algorithms and Techniques 161
A.3 Individual Failure Modes 162
A.4 Multi-Class Classification 163
A.5 Ensembling the Decisions of ML Models 163
A.6 Discussion 170
Appendix B: Netdata Complete Metrics Set 171
Appendix C: Netdata Selected Metrics Set 183
Appendix D: Failure Detectors 189

List of Figures

1.1 Framework to Enable Advanced OFP 6

2.1 Fault-Error-Failure-Fault, adapted from [Avizienis et al., 2004] . . 12
2.2 Online Failure Prediction (OFP) Illustration 13
2.3 Failures, Time-to-Recover (TTR), Time-to-Failure (TTF), and Time-Between-Failures (TBF) 14
2.4 OFP Problem Statement, adapted from [Salfner et al., 2010] . . . 15
2.5 Fault Injection Environment, adapted from [Hsueh et al., 1997] . 16
2.6 Machine-code Fault Injector 18
2.7 Generic ML Pipeline, adapted from [Duboue, 2020] 24
2.8 Machine Learning Types 27
2.9 Sliding Windows-Based Time Series Analysis 34
2.10 Adversarial Example [Goodfellow et al., 2015] 35

3.1 Experimental Process 50
3.2 Performance Isolation Process 51
3.3 Data Generation Process 56
3.4 Propheticus High-Level Architecture/Workflow 62

4.1 Methodology to Create Predictors for OFP 68
4.2 Experiment Labeling [Salfner et al., 2010] for $\Delta t_l = 20$ and $\Delta t_p = 30$ 70
4.3 Experiment-wise Leave-one-out Cross-Validation (ELOOCV) . . . 74
4.4 Experiment-wise Failure Prediction (4 alerts in the last 5 seconds) 76

5.1 General Architecture of the Framework 80
5.2 Benchmarking Procedure 85

6.1 *cpu* Workload Time-to-Failure (TTF) 101
6.2 *cpu* Workload [20, 40] Features Means 103
6.3 Classes Distribution 104
6.4 CPU workload [20, 40] Heat Map 105
6.5 SVM w/ RBF, *cpu*, [40, 40] 108
6.6 Decision Tree (DT), *cpu*, [20, 40] 108
6.7 DT, *cpu*, [20, 40] 109
6.8 XGBoost, *cpu*, [20, 40] 109
6.9 DT, Comb., [20, 40] 110
6.10 XGBoost, Comb., [20, 40] 110
6.11 F_2-score and $\Delta t_l, \Delta t_p$ 111
6.12 XGBoost, Best, [30, 200] 111
6.13 Decision Tree Model for Memory Failures 113
6.14 XGB., CV [20, 40] 117

6.15 XGB., Experiment-wise Leave-one-out Cross-Validation (ELOOCV) [20, 40] 117
6.16 Hang, [30, 80] 120
6.17 CPU/Exec, [60, 20] 120
6.18 Number of Samples per L_∞ Perturbation (up to 0.3) 133

A.3.1 Crash: Support Vector Machine (SVM), Correlation 163
A.3.2 Hang: SVM 163
A.5.1 Experimental Process 164
A.5.2 Base Learners Selection 166
A.5.3 Best Soft Voting Ensemble Graph 168
A.5.4 Best Soft Voting Ensemble Venn 169
A.5.5 Best Soft Voting Ensemble Prediction Contributions Venn 169

List of Tables

2.1 Error Types 32

5.1 Recommend Metrics by Scenario 82

6.1 Workloads - stress-ng [Ubuntu, ndf] 93
6.2 Isolation Experiments 94
6.3 Experiments Isolation 95
6.4 Fault Types 97
6.5 Failure Modes 98
6.6 Experiments per Workload 98
6.7 Software Implemented Fault Injection (SWIFI) Summary 100
6.8 Techniques Used in the Experiments 106
6.9 Algorithms' Hyperparameters 106
6.10 F_2-score per lead-time (40 sec. prediction-window) 111
6.11 F_2-score per sliding-window 112
6.12 Techniques Used in the Experiments 118
6.13 Algorithms' Hyperparameters 119
6.14 Experiments Distribution per Failure Class 120
6.15 Experiment-wise Prediction Results 122
6.16 5-seed Experiment-wise Summary 122
6.17 Techniques Used in the Experiments 126
6.18 Algorithms' Hyperparameters 126
6.19 Rank per Failure Mode/Scenario 128
6.20 Linux Unrobust Model vs Adversarial Samples (max $L_\infty = 0.3$) . 131

A.5.1Algorithms' Hyperparameters 165
A.5.2Diversity Metrics per Ensemble (Soft Voting) 170

Acronyms

ADASYN	Adaptive Synthetic.
AHP	Analytic Hierarchy Process.
AI	Artificial Intelligence.
ARIMA	AutoRegressive Integrated Moving Average.
CART	Classification and Regression Trees.
CLI	Command Line Interface.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.
CS	Computer Science.
DBSCAN	Density-Based Spatial Clustering of Applications With Noise.
DNN	Deep Neural Network.
DT	Decision Tree.
EDA	Exploratory Data Analysis.
ELOOCV	Experiment-wise Leave-one-out Cross-Validation.
G-SWFIT	Generic Software Fault Injection Technique.
GAN	Generative Adversarial Network.
GARCH	Generalized AutoRegressive Conditional Heteroskedasticity.
GB	Gradient Boosting.
HC	Hierarchical Clustering.
HPC	High-Performance Computing.
HSFI	Hybrid Software Fault Injection.
ID3	Iterative Dichotomizer 3.
IRQ	Interrupt Request.
k-NN	k Nearest Neighbors.
LBFGS	Limited-memory Broyden–Fletcher–Goldfarb–Shanno.
LDA	Linear Discriminant Analysis.
LTS	Long-term Support.
MAE	Mean Absolute Error.

ML	Machine Learning.
MLP	Multilayer Perceptron.
MSE	Mean Squared Error.
MTBF	Mean-Time-Between-Failures.
MTTF	Mean-Time-to-Failures.
MTTR	Mean-Time-to-Repair.
NFV	Network Function Virtualization.
NLP	Natural Language Processing.
NN	Neural Network.
NUMA	Non-Uniform Memory Access.
ODC	Orthogonal Defect Classification.
OFP	Online Failure Prediction.
OS	Operating System.
PCA	Principal Component Analysis.
RBF	Radial Basis Function.
RCU	Ready-Copy-Update.
ReLU	Rectified Linear Units.
RF	Random Forest.
RFE	Recursive Feature Elimination.
RMSE	Root Mean Square Error.
RNN	Recurrent Neural Network.
ROC	Receiver Operating Characteristics.
RSS	Resident Set Size.
SAT	Static Analysis Tool.
SMOTE	Synthetic Minority Over-sampling Technique.
SSD	Solid-State Drive.
SSE	Sum of Squared Errors.
SVM	Support Vector Machine.
SWIFI	Software Implemented Fault Injection.
TBF	Time-Between-Failures.
TCL	Tool Command Language.
TTF	Time-to-Failure.
TTR	Time-to-Recover.
VAE	Variational Autoencoder.
VM	Virtual Machine.

Chapter 1

Introduction

Critical and sensitive tasks are executed through complex software systems on a daily basis. Issues or failures on such systems can easily lead to considerable financial losses or even loss of lives. At the same time, software size and complexity have been growing considerably, reaching proportions that render traditional validation techniques impractical. As a result, it has become virtually impossible to detect all software faults before deployment. Software faults are in fact one of the main causes of system outages [Dhanalaxmi et al., 2015].

Residual faults, which escaped the testing processes, may eventually lead to failures that impact availability and reliability and thus compromise the supported business processes. Depending on the purpose of the system, such faults, and consequent failures, may incur considerable risk or cost, either due to the recovery mechanisms or the resulting system interruption or corruption. In recent years, system outages have cost significant amounts to numerous companies (e.g., Amazon outage in 2018 reportedly cost around $100 million [Ed Targett, 2018] and Facebook outage in 2019 approximately $90 million [Brown, 2019]) and software faults have also been identified as a major contributor to incidents leading to loss of lives (e.g., Boeing 737 Max crashes [McFall-Johnsen, 2020]). Various techniques have been developed with the purpose of avoiding and detecting such faults, including coding practices, approaches for testing, and models for reliability characterization [Avizienis et al., 2004].

Online Failure Prediction (OFP) is a fault-tolerance technique that intends to mitigate the potential effects of residual faults, by using past data and the current system state for predicting the potential occurrence of failures [Salfner et al., 2010]. This allows taking preemptive measures to avoid such failures or mitigate their consequences. By being able to predict failures, complex infrastructures can reduce the associated risks, while at the same time improving dependability attributes, namely availability and reliability. In practice, even large and highly complex infrastructures such as High-Performance Computing (HPC) systems (and even exascale platforms) can benefit from failure prediction mechanisms [Bouguerra et al., 2013]. Altogether, the systems that can benefit the most from these advances are large servers, including both cloud and virtualized systems, where the costs and risks associated with a failure are not negligible. More precisely, systems where existing fault-tolerance and failover mechanisms are not sufficient (e.g., computation lost at the time of failure must be redone), as well as complex systems where such techniques (e.g., redundancy) are not applicable.

Despite the great potential of OFP, it is still not widely implemented. One reason is that failures are rare events and thus collecting enough data to develop accurate failure predictors is a complex (and expensive) endeavor. Moreover, considering the current rate at which software evolves, existing failure data may quickly become deprecated. Another reason for the limited use of OFP is that properly creating representative and accurate predictive models is not trivial and requires expert knowledge on various interdisciplinary subjects. In fact, effectively developing OFP solutions requires an adequate selection of the most suitable models for a particular system installation. This calls for a rigorous assessment of alternative solutions using appropriate metrics and techniques, and their comparison using adequate datasets and procedures.

1.1 Problem Statement

Failure prediction is a technique that relies on data concerning past failures to predict failures in the future. OFP is a specific case of failure prediction that also takes the current state of the system into account to predict incoming failures [Salfner et al., 2010]. OFP relies on the premise that the system will exhibit some out-of-norm behavior before a failure event, which can be interpreted as a symptom. Using such predictions, it is possible to take preemptive measures to avoid, or at least mitigate, the consequences of a failure. Due to its complexity and the interdisciplinarity of the knowledge required, developing accurate predictive models for OFP remains an open issue.

A significantly growing investment into the ability to extract knowledge from data has been observed in recent years. Large companies such as Google, Amazon, and Netflix, invest millions to improve their businesses (e.g., recommendations, search heuristics), while others, such as PayPal, Symantec, and Tesla, use it for more critical applications (e.g., fraud detection, cyber-security, autonomous driving) [Siegel, 2016; Elliott, 2019]. As a result of this demand for new and better algorithms, Machine Learning (ML), a sub-field of Artificial Intelligence (AI), has expanded considerably. Although it has been around for several years, the recent technological developments have allowed the use of ML algorithms on significantly larger and more complex datasets, bringing it back to the spotlight.

ML algorithms have shown their ability to adapt and extract knowledge in a variety of complex problems. One of the main reasons is that they are able to find intricate patterns in the data and learn from them, often without assumptions about the underlying model. In practice, they can be used to make predictions on new unseen data based on what was previously learned. Although there are many different algorithms, it has been proven that they perform differently depending on the nature of the problem. This led to the *No Free Lunch Theorem* [Wolpert and Macready, 1997], which states that, on average, there is no algorithm that is better for all instances of all classes of problems. This way, different algorithms and techniques should be considered and experimented, to find the one that best models the problem. However, such an approach is highly complex, as many of the tasks required in the process, such as the feature selection and the choice of the parameters and hyperparameters of the algorithms, are themselves optimization problems. In fact, a thorough ML approach is complex and problem-dependent and thus requires a deep understanding of both the problem and its data to select the most adequate algorithms and techniques. Furthermore, real-world problems often present certain characteristics that need to be taken into account (e.g., imbalanced data, high or low dimensionality), and properly assessing the performance of the models is also not trivial. All these considerations lead to the necessity of mastering both theory and practice, otherwise, the results and subsequent conclusions may be compromised.

Developing accurate predictive models for OFP requires not only accurate tuning but also assessing and comparing multiple alternative solutions using appropriate metrics and adequate procedures. To make fair comparisons, this process must be well-defined, such that the assessment of the performance of the predictors provides confidence on how the results will hold in the operational scenario. Due to the lack of such a procedure, existing works use ad-hoc approaches, consider a diverse set of metrics without thorough consideration of the purpose/needs of the system, and neither statistically validate differences nor assess how sensitive the models are to minor variations in the data (similar to what occurs in the real-world).

Different approaches based on ML have already been proposed for OFP [Salfner et al., 2010]. Various sources of input data (e.g., symptoms monitoring, detected error reporting) can be used, each with different techniques that better exploit the information available. However, most existing works on OFP are limited, as they are fairly dependent on the subject of a particular study. The main problem is that, due to the inherent complexity of both OFP and ML, most works focus on a small set of prediction algorithms and techniques (usually based on what has previously been used on a similar problem), in a very specific context, predicting specific failures, and using only limited fractions of the datasets. As ML became widely adopted, various platforms that abstract some of the technical details (e.g., Weka [Eibe et al., 2016]) have been developed. Still, most cannot be easily customized or extended, contain small or limited libraries, have small communities and consequently evolve slowly, or abstract on a lower level and still require significant coding. In fact, albeit comprehensive tools such as Weka are often used, there are many researchers that resort to lower-level libraries, such as scikit-learn [Pedregosa et al., 2011]. This suggests that, although such tools are adequate for certain purposes, they are not flexible or easily adaptable for many others.

A prevalent challenge to the widespread adoption of OFP is the scarcity of failure data, especially for complex systems (such as Operating Systems (OSs)). While OFP can be used to predict the failures of individual components, system-level OFP requires a more complete solution, able to predict failures of the system as a whole. However, failures are rare and thus failure data (required to create predictive models) is typically not available. Collecting failure from real systems would take years (due to the reliability of modern systems) and by then would already be outdated. There are various initiatives aiming at building failure data repositories (e.g., the Computer Failure Data Repository [Usenix and University, nd], Los Alamos National Laboratory [Laboratory, nd]), but using such datasets is not enough, as they do not allow taking into consideration the system (and workload) where the predictors will run. Additionally, OFP requires the sequential/continuous monitoring of the system both under non-failure (i.e., baseline) and failure-prone states (i.e., before the failure occurs), which are typically not considered in those datasets (e.g., most provide only the failure logs).

Fault injection has been accepted as a viable alternative to generate failure data in a reasonable amount of time (e.g., [Irrera and Vieira, 2015; Cotroneo et al., 2019]). In practice, fault injection is used to inject realistic programming faults, which allows studying the response of the system to the presence of faults. Monitoring the target system during the experiments allows capturing its behavior between fault activation and subsequent (potential) failure. However, proper fault injectors are usually complex pieces of software that are difficult to implement and use. Even with a viable fault injector, considerable effort/knowledge is still required to set it up (e.g., many require recompiling the system/kernel) and to implement the whole experimental process (e.g., fault load, failure modes/monitors). Moreover, conducting a fault injection campaign on a complex system requires executing thousands of experiments to achieve statistical relevance. Recent technological developments have led to an increase in computational power, and various techniques have been developed to leverage it. Still, implementing a testbed that can take advantage of the resources available without influencing the results is not trivial. To expedite the experimental process and reduce costs researchers often use modern techniques (e.g., hyperthreading) to run multiple experiments simultaneously. This relies on a premise of non-interference, that is, simultaneous execution should not alter the behavior of the individual experiments. While some types of isolation are easier to attain (e.g., *software isolation*, the corruption or misbehavior of one experiment should not influence other experiments) others are quite difficult (e.g., *performance isolation*, where executing one or multiple experiments simultaneously should lead to similar results). This poses a challenge to the repeatability and validity of the experiments. As documentation, guidelines, and examples are not usually available, the process requires significant effort and expertise to identify all the relevant attributes, requirements, and implementation solutions.

Due to all the open issues and knowledge/expertise required, most recent work on OFP focuses on smaller components (e.g., hard-drives [Zhang et al., 2020]) with a shorter lifespan (for which it is possible to find failure datasets). The fact is that fault injection, ML, and OFP are all complex interdisciplinary subjects whose combination poses considerable challenges, especially because there is little to no related body of literature. Current related work focuses on each of the techniques individually, and it is up to the researcher to find out how to use and combine them to create failure predictors. As no thorough solutions, methodologies, or procedures are available, advanced research on OFP for complex systems, which relies on both fault injection and advanced ML, has become stale to non-existent. Furthermore, due to the lack of well-defined and established procedures, the research that has been made so far on the subject (e.g., [Irrera and Vieira, 2014]) relied on standard ML techniques which are not adequate for the specific characteristics of the problem, and that ultimately lead to biased and unrepresentative observations.

1.2 Drivers and Framework

This work is mainly motivated by the pervasiveness and development of ML, the growing complexity of modern software, and the unexplored potential of OFP as a solution for increasing the reliability and availability of complex systems. It focuses on identifying and characterizing the main challenges and limitations to the use of OFP and searching, assessing, and devising potential solutions and procedures. Overall, the main drivers of this work are:

- **Online Failure Prediction (OFP) on specific systems and workloads**
  Failures are rare and, as a result, failure data are typically not available. Although there have been some efforts into creating public datasets, failures are dependent on the specific system where the predictors will operate. One of the main drivers of this work is being able to study and create representative failure predictors for a specific target system and workload.
- **Machine Learning (ML) to create accurate failure predictors**
  Due to the complexity of modern systems, the ability to characterize and predict incoming failures based on the current system state is not certain. Another prevalent driver of this work is to assess and study the viability of using both classic and state-of-the-art ML methods to create accurate failure predictors and the impact and influence of the different techniques for OFP.
- **Online Failure Prediction (OFP) parameters and characteristics**
  OFP has various parameters (e.g., *lead-time*, *prediction-window*) and presents certain characteristics (e.g., time series with failures at the 'end') which influence the predictors. The last main driver of this work is to thoroughly explore the problem of OFP, and how its parameters influence the creation of accurate failure predictors. Additionally, it also includes studying how the specific characteristics of OFP should be taken into consideration when creating predictive models.

This work advances the state of the art on OFP by reducing the existing gap in the literature concerning the *use of fault injection and ML to create representative and accurate predictive models for OFP*. To overcome the existing challenges and limitations of OFP, a framework comprised of five interconnected elements is proposed, as illustrated in *Figure 1.1*. It is divided into two groups, *techniques and artifacts* and *procedures and methodologies*. Each contribution within these groups focuses on a different aspect of the problem, namely: 1) configuring and deploying a *testbed for dependability experiments*, 2) using *fault injection techniques to generate failure data*, 3) exploring the problem and *developing accurate predictive models*, 4) combining the use of *fault injection to generate failure data to create predictive models for OFP*, and 5) benchmarking and properly *assessing and comparing failure predictors*. Using the novel techniques and artifacts and following the proposed methodologies and procedures will allow researchers to explore the use of OFP in complex systems, further advancing the state of the art.

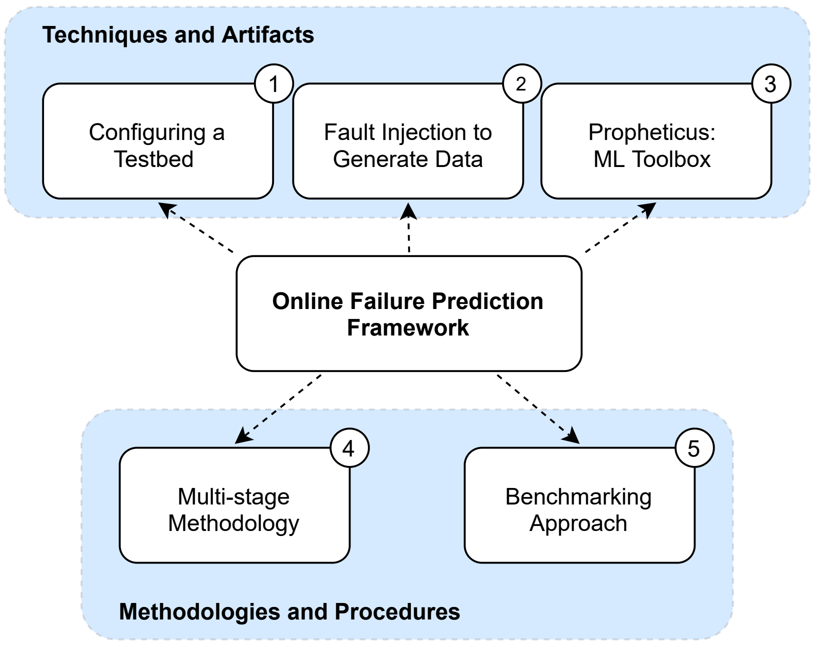


Figure 1.1: Framework to Enable Advanced OFP

To demonstrate how it can be used in practice, an instantiation of the proposed framework is presented, targeting an up-to-date Linux kernel. It comprises an extensive fault injection campaign to generate failure data, considering multiple workloads representing different usage scenarios, several fault types, and various failure modes. Afterward, a detailed analysis of the viability and performance of several ML methods, ranging from classic to state-of-the-art approaches, is conducted. Results demonstrate that not only is it possible to create accurate failure predictors but that the proposed techniques are essential to guide the process and assure a representative and sound experimental process.

1.3 Contributions

The work presented in this thesis attempts to overcome the current challenges and limitations to the widespread use of OFP, providing a **comprehensive multi-part framework and proposing techniques and tools to address the most pressing open issues, from generating realistic failure data to developing and benchmarking predictive solutions**. This work fills an existing void in the literature on combining advanced dependability and ML techniques to create accurate failure predictors for modern complex systems.

Configuring and Deploying Testbeds
Our contribution dwells on how to design and implement a proper testbed for experiment-based research. It focuses on assuring that the results of the experiments are as consistent and repeatable as possible, regardless of external influences, while leveraging the computational power available. It provides guidelines and reflections on the various techniques and steps required to achieve this. The main contribution is a comprehensive overview of the various concerns and re-

quirements of a testbed for generating data for OFP and clear guidelines on how to implement it on the Linux OS. This allows researchers to properly leverage their computational resources and expedite the experimental process, as well as reduce hardware costs, without compromising the validity of the results.

Fault Injection to Generate Failure Data
Our contribution includes overviewing and providing guidelines on how to use fault injection to generate representative failure data to support the development of predictive models for OFP. The main advance is a well-defined process with a set of guidelines and considerations that should be taken into account when devising a fault injection campaign to generate data for OFP on an up-to-date complex system. Combined with the previous contribution (configuring and deploying testbeds), they provide all the necessary guidelines and processes to facilitate researchers on using fault injection to generate failure data in a timely manner.

Propheticus: Machine Learning (ML) Toolbox
We propose Propheticus, a ML toolbox that attempts to abstract the complexity of ML whilst being easy to use and accommodate the needs of the users in a dependability research context. It includes functionalities for all the steps required in a ML approach, from data analysis to model assessment and comparison. Due to the variety of tasks and problems within dependability research, it is focused on flexibility, using a plugin-based system that easily allows including new ML techniques (from data splitting techniques to algorithms and performance metrics). Additionally, Propheticus implements a hook-based system, which allows the user to seamlessly implement their own code and integrate it with the different functions of the system without changing the source code. Ultimately, the main contribution is a tool that can be used and adapted to explore and assess the performance of the different ML techniques on a given problem, such as creating failure predictors using data generated through fault injection.

Methodology for Developing Predictive Models
We propose a six-stage methodology for developing predictive models for OFP that takes into consideration the specificity of using fault injection to generate failure data. It highlights the need to consider the specific characteristics of such data (e.g., repeated controlled experiments, where the failures only occur at the end of the experiment) with those of OFP (e.g., time series) to train and assess the performance of predictive models. It also focuses on how to explore and process the failure data, such as identifying and refining the failure classes, and all the potential iterations and feedback loops when working with a new, unexplored, dataset. This contribution addresses a major open issue and gap in the literature, on how to combine these three interdisciplinary subjects. This work intrinsically leverages the previous contributions using fault injection to generate failure data and Propheticus to create predictive models.

Benchmarking Predictive Models
We propose a conceptual framework for benchmarking failure predictors, ensuring a sound assessment and comparison of the various solutions. It is comprised of

several detailed phases, considering various concepts such as scenarios (a realistic situation on failure prediction that depends on the criticality of the system, which defines the level of dependability that should be satisfied at the cost of mitigating predicted failures) and the robustness of the predictors to small variations in the data (which gives credibility to the representativeness of the results obtained in the benchmarking process). This contribution complements the methodology previously described (which follows a more exploratory approach) by providing the means to conduct a thorough and systematic comparison between alternative solutions.

Experimental Evaluation
An experimental evaluation of the various contributions is presented. This comprises an extensive fault injection campaign on a system based on an up-to-date Linux kernel, considering different workloads representing different usage scenarios, several fault types, and various failure modes. The data are then thoroughly explored and a comprehensive study is conducted on the use of ML techniques, from traditional to state-of-the-art algorithms, and their applicability to OFP. To illustrate how the choice of the best solution should consider the needs of the user, the performance of the various models is also benchmarked using metrics that take into consideration the technical needs of the system where they will operate. Results demonstrate that not only it is possible to create accurate failure predictors, but also that the techniques, artifacts, methodologies, and procedures proposed are essential to guide the process and assure a representative and sound experimental process.

1.4 Outline of the Document

The remainder of the document is structured as follows.

Chapter 2 provides a background revision on dependability, focusing on OFP, fault injection, testbeds, and benchmarking. It also includes a comprehensive revision of ML concepts, from the ML workflow (which includes several items such as data analysis and preparation, learning types and algorithms, and evaluation) to more specific topics such as time series and adversarial ML. A thorough overview of important related work in the literature is also included in this chapter.

Chapter 3 focuses on techniques and artifacts to support OFP. It covers how to properly implement a testbed for dependability experiments, how to use fault injection to generate failure data to support OFP, and a ML toolbox for exploring and creating predictive models.

Chapter 4 proposes a multi-stage methodology for exploring and developing predictive models for OFP, from generating and processing the data, to identifying and refining the failure classes of the problem and ultimately deploying the models, while taking into consideration the specific characteristics of both fault injection and OFP.

Chapter 5 introduces a benchmarking approach to properly assess and compare ML solutions for OFP, with several considerations such as scenarios (realistic situations of failure prediction that depend on the criticality of the system) and the most adequate corresponding metrics, how to properly compare alternative solutions, and the need to tolerate small unrepresentative variations in the data.

Chapter 6 presents the experimental evaluation of the framework, including an instantiation of all the contributions, from configuring and deploying a testbed and using fault injection, to applying the methodology to create accurate failure predictors and properly benchmarking them.

Chapter 7 closes the thesis by presenting the final remarks and conclusions and putting forward ideas for future research topics and directions.

Appendix A presents an exploratory study on using the proposed techniques to develop failure predictors for the Windows OS. *Appendix B* presents the complete set of metrics collected during the fault injection campaign targeting the Linux OS and *Appendix C* details the subset of those metrics that were used to develop the predictive models. Finally, *Appendix D* details the various failure modes considered for the Linux dataset.

Chapter 2

Background and Related Work

Advancing the state of the art on OFP requires mastering several concepts and techniques from two interdisciplinary research fields: *dependability* and *artificial intelligence*, more precisely, *machine learning.* This chapter introduces the most relevant background concepts on *dependability*, with a focus on *Online Failure Prediction (OFP)* and *Machine Learning (ML)*, in sections *2.1* and *2.2*, respectively. *Section 2.3* overviews literature related to the work presented in this thesis.

2.1 Dependability

The growing complexity of software makes it difficult or even impossible to detect all faults before deployment. Although a significant effort is usually put into testing the products to remove as many faults as possible, '*testing shows the presence, not the absence of bugs*', as once stated by Edsger W. Dijkstra. Such residual faults, that escaped the testing processes, eventually lead to failures that impact the availability and reliability of systems and thus compromise the supported business processes. In fact, software faults have been recurrently recognized as one of the main causes of system outages [Gray, 1986; Oppenheimer et al., 2003; Dhanalaxmi et al., 2015].

A *system* is a complex concept, which has multiple definitions. Yet, within the dependability community, one of the most accepted is that it is '*an entity that interacts with other entities, [...], including hardware, software, humans, and the physical world with its natural phenomena*' [Avizienis et al., 2004]. The frontier between the system and its environment is known as the *system boundary.* The service provided by the system (i.e., its behavior as perceived externally) is delivered through *service interfaces*, which are the parts of the system boundary where service delivery takes place. The part of its state that is perceived at the interface is known as the *external state*, and the remaining is the *internal state.* The delivered service is, in fact, the product of a sequence of external states of the system that is perceivable at its interface [Avizienis et al., 2004].

Following the taxonomy used by Avizienis et al. [Avizienis et al., 2004], when the service correctly implements the system function it is considered a *correct service* delivery. The transition from the correct service into an *incorrect service* is known as a *service failure.* The duration of the incorrect service is known as a *service outage*, and its recovery is known as *service restoration.* There are multiple ways the service can deviate from its expected behavior, commonly known as service failure modes (e.g., content, timing).

A service failure is due to the fact that the state of the system deviated from the correct one. This deviation is known as an *error*. However, an error does not necessarily lead to a failure, as it is not considered a failure until it reaches the service interface or is detected by an external user. In fact, the system can amass errors without compromising its service delivery, causing a partial failure, or even running in degraded mode. Besides potentially leading to failure, an error may cause out-of-norm behavior of the system parameters as a side effect, which are known as *symptoms* [Salfner et al., 2010].

Faults, which can be either internal (i.e., originated within the system boundaries) or external (i.e., originated outside the system but propagated by interaction or interference), are the hypothesized cause of an error. In most cases, faults are regularly dormant, until they become active and cause an incorrect system state, which is why errors are known as the manifestation of faults [Salfner et al., 2010]. Faults can have different classifications, such as transient, intermittent, and permanent faults [Siewiorek and Swarz, 1998; Avizienis et al., 2004]. An illustration of the relationship between fault, error, and failure (and subsequent external fault in other system) can be seen in *Figure 2.1*.

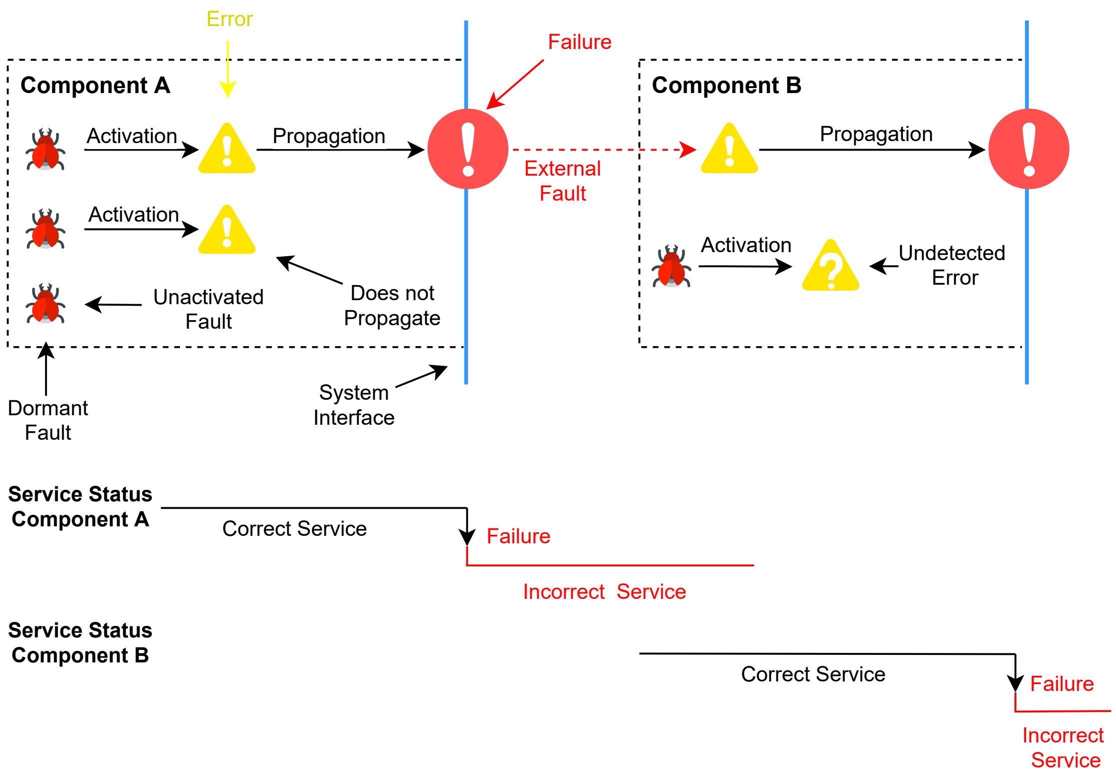


Figure 2.1: Fault-Error-Failure-Fault, adapted from [Avizienis et al., 2004]

Dependability is '*the ability to deliver service that can justifiably be trusted*', as well as '*the ability to avoid service failures that are more frequent and more severe than is acceptable*' [Avizienis et al., 2004]. It is in fact, a composite concept that comprises different attributes, such as availability, reliability, safety, integrity, and maintainability. Multiple techniques have been developed with the purpose of avoiding and detecting faults. More precisely, they can be divided into four major groups [Avizienis et al., 2004]:

- **Fault prevention** – part of general engineering, attempt to prevent the existence of faults in the system (e.g., improvement of development processes for both software and hardware);
- **Fault tolerance** – recognizing the inevitable existence of faults that will lead to errors, are aimed at avoiding failures (e.g., use of redundancy of systems, mitigating the effects of failures). Additionally, such techniques can focus on detecting and handling errors (e.g., roll-back/forward) and on fault detection (e.g., memory leaks);
- **Fault removal** – focus on removing faults, either during the development life-cycle (e.g., verification and validation) or at runtime (e.g., symbolic execution and online testing);
- **Fault forecasting** – attempt to predict the consequences of faults by evaluating (qualitatively or quantitatively) the system behavior concerning fault occurrence and activation, such as using system modeling (e.g., reliability growth models).

2.1.1 Online Failure Prediction (OFP)

Failure prediction (also known as reliability modeling) relies on information regarding failures that happened in the past to predict future failures, making it possible to estimate indicators such as Mean-Time-to-Failures (MTTF) and Mean-Time-Between-Failures (MTBF). Yet, these techniques consider solely past data, not taking into account the current state of the system, thus providing limited prediction accuracy. To address this, approaches that take into the system state at the time of the prediction started to appear around 1997 [Wolski et al., 1997] and became known as *Online Failure Prediction (OFP)*. As depicted in *Figure 2.2*, OFP intends to mitigate the potential effects of residual faults, by using past data and the current system state to predict the potential occurrence of failures in the near future [Salfner et al., 2010].

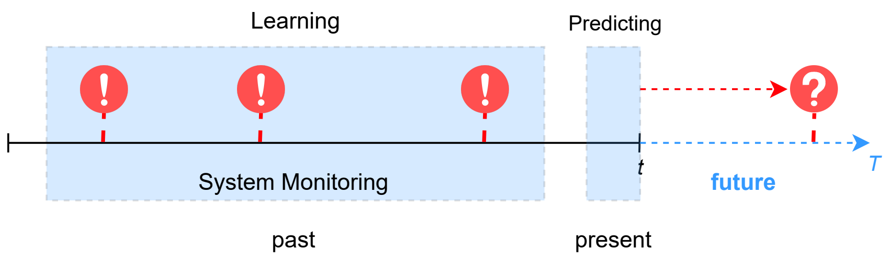


Figure 2.2: Online Failure Prediction (OFP) Illustration

By using OFP to predict incoming failures, one can take preemptive measures to avoid such failures or mitigate their consequences [Salfner et al., 2005]. In the eventuality of not being possible to avoid them, such information can be used to initiate recovery mechanisms, expediting system recovery. As illustrated in *Figure 2.3*, anticipating recovery mechanisms reduces Mean-Time-to-Repair (MTTR), thus improving availability, whilst avoiding failures increases MTBF, increasing reliability [Irrera, 2016]. The improvement of these attributes directly influences the trustworthiness of the system, by providing more reliable and available systems.

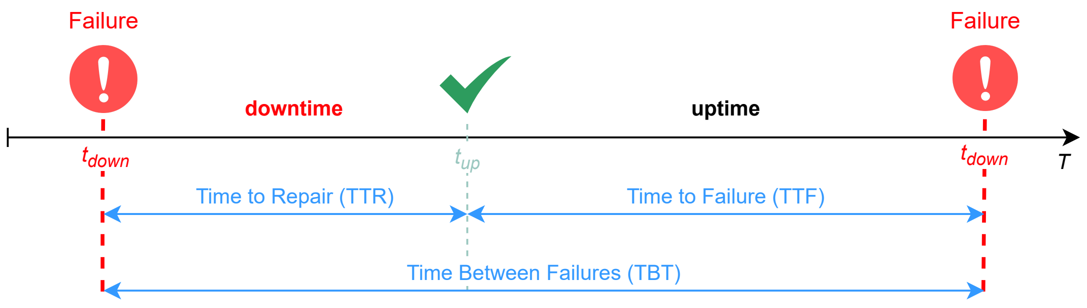


Figure 2.3: Failures, Time-to-Recover (TTR), Time-to-Failure (TTF), and Time-Between-Failures (TBF)

Multiple approaches have already been proposed for OFP, as surveyed by Salfner et al. [Salfner et al., 2010]. By studying the literature, the authors proposed a taxonomy based on the fault-error-failure model. The authors consider that the system has four possible states (i.e., *fault, undetected error, detected error,* and *failure*) in the path to failure, which can also be accompanied by *symptoms* (an out-of-norm behavior of the system parameters caused by errors). The existing approaches can be grouped based on the type of input data they use [Salfner et al., 2010]:

- **Undetected errors** – can be found through auditing, which are techniques that analyze the system to check if it is in an incorrect state;
- **Detected errors** – after an error detector detects an incorrect state it records that information into a log file. These logs are then processed and analyzed to predict incoming failures;
- **Symptoms** – are side effects of errors and can be identified by monitoring the system parameters for out-of-norm behavior. Some types of errors affect the system even before the failure occurs (sometimes referred to as service degradation);
- **Failures** – can be made visible by tracking mechanisms, the idea is to make conclusions about upcoming failures based on the occurrence of previous ones.

Amongst the four, one of the most promising approaches is *symptoms monitoring* (the work conducted in this thesis is focused on this approach). Various techniques already exist that take advantage of analyzing the system parameters. One of the most known is *software rejuvenation*, which is a proactive technique that reduces the probability of failures through *software aging* (a phenomenon where the performance of the system degrades over time leading to failure).

The problem of OFP was formalized as a decision process for predicting at time t if a failure is going to occur within a precise time, called *lead-time*, Δt_l. Such prediction can be valid for a given time window, called *prediction-window* Δt_p [Salfner et al., 2010]. In practice, at time t, the model (trained using previous data on failures) should predict if a failure is going to occur in the interval $[t + \Delta t_l, t + \Delta t_l + \Delta t_p]$. An illustration of the problem can be seen in *Figure 2.4*. The width of the Δt_l defines how far ahead the failure is to be predicted, and represents the ideal lead time to avoid or initiate repair mechanisms, and must be greater than the minimal *warning-time*, Δt_w, otherwise, there will not be enough time to react. That prediction is valid for the period Δt_p, which if it is too large will most likely increase the prediction ability, yet it may become useless, as it is not clear when the failure will occur [Irrera, 2016]. For the prediction, some set of data previous to the instant t, Δt_d, can be used to account for the evolution of the parameters. The purpose of the models can be to either predict if a failure will occur or a continuous measure, indicating how failure-prone the system is.

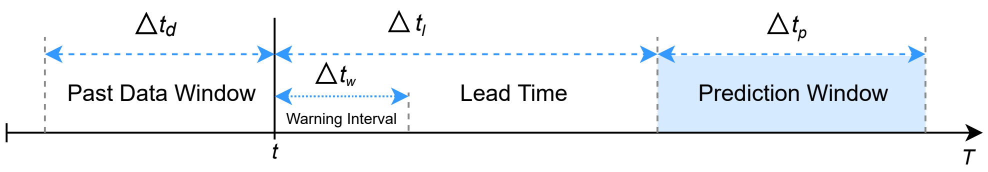


Figure 2.4: OFP Problem Statement, adapted from [Salfner et al., 2010]

One of the problems with OFP and one of the main reasons why it is not widely used is that it requires a significant amount of failure data. However, due to the reliability of modern systems, failures are rare events and thus collecting data for training and testing new methods is a complex (and expensive) endeavor. To address this, various studies have been conducted on using fault injection to generate representative failure data [Durães and Madeira, 2006; Irrera and Vieira, 2015; Irrera et al., 2015].

2.1.2 Fault Injection

Fault injection is a technique that intentionally introduces faults in the system in an attempt to mimic real faults [Arlat et al., 1990], with the purpose of assessing the behavior of the system. With it, it is possible to assess the impact that residual faults have on the system (e.g., fault tolerance, dependability validation) [Irrera, 2016]. Fault injection can be used to mimic both *hardware* (e.g., bit-flip) and *software* (e.g., missing variable assignments) faults, and can also be injected through hardware or software (also known as Software Implemented Fault Injection (SWIFI)) [Hsueh et al., 1997]. Yet, due to the recent ever-growing complexity of software, in the last few years, the focus of research on fault injection has been on software faults injected through software. It is possible to identify various classes, such as *injection of data errors* (e.g., corrupting memory or registers), *interface error injection* (e.g., corrupting input or output values at the component interface), and *injection of code changes* (e.g., introducing wrong code to emulate software faults) [Natella et al., 2016]. As the purpose of this thesis is to predict failures caused by residual faults, only the latter is considered. Note that, previous empirical studies have demonstrated that the injection of code changes can realistically emulate software faults [Daran and Thevenod-Fosse, 1996; Andrews et al., 2005].

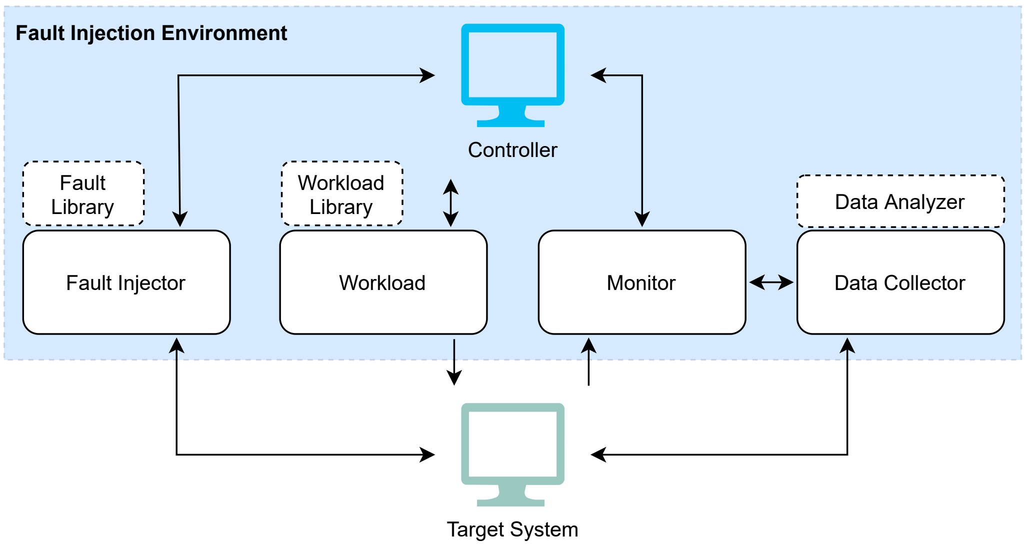


Figure 2.5: Fault Injection Environment, adapted from [Hsueh et al., 1997]

A fault injection approach includes two main components: the fault injection tool and the target system [Hsueh et al., 1997]. An illustration of the common structure can be seen in *Figure 2.5*, which contains a *controller* (controls the experiment), a *fault injector* (the tool that actually injects the faults), a *fault library* (specifies which, where, and when to inject faults), a *monitoring system* (monitors the system to detect the effects of the faults, which are then collected by the data collector and data analyzer), and a *workload generator* (to exercise the system).

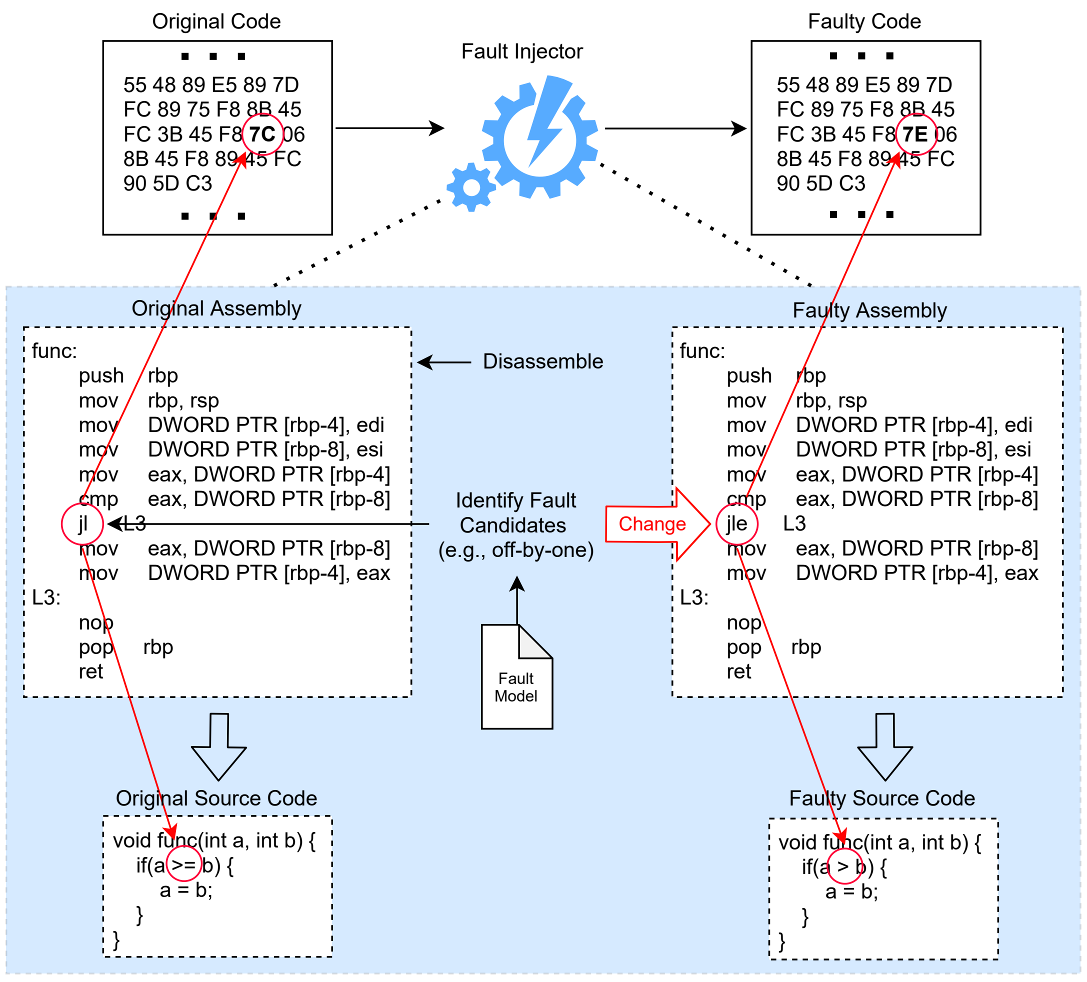
The emulation of software faults presents several challenges, such as *what*, *where*, and *when* to inject faults. Additionally, the very definition of the fault model, which comprises the set of faults to inject, is complex. One of the first proposed models was the Orthogonal Defect Classification (ODC) [Chillarege, 1996], which defines software defects and triggers. In short, it classifies the way a programmer would correct the faults: *assignment, checking, interface, timing/serialization, algorithm, function, build/package/merge,* and *documentation*. Although this approach was used to guide fault injection research, it was later pointed out that it was not adequate for fault injection, as it is not organized by how the faults should be emulated [Durães and Madeira, 2006]. *Duraes and Madeira* then proposed an extension to the ODC based on the premise that a defect is one or more programming language constructs that are either *missing, wrong*, or in *excess.*

Faults can be injected at different levels. Injecting faults at the *source-code* level is simpler and can be more precise as the whole context is preserved. However, on many occasions, the source-code is not available and this approach does not scale well to large code-bases [Van Der Kouwe and Tanenbaum, 2016]. It is also possible to inject faults at the *binary/machine-code* level, which is easier to port, faster to execute, and scales better [Durães and Madeira, 2006]. However, this approach is less accurate as it loses contextual information [Cotroneo et al., 2012]. In some cases, it is also possible to inject faults at an *intermediate-code* level (e.g., LLVM intermediate representation), which provides a balance between source- and machine-code level fault injection.

The use of fault injection does not come without its risks, one of which is how to properly assure that the injected faults are indeed representative of faults that escaped the testing phases (i.e., residual faults). Although this does not have an explicit and final answer, it has been addressed in several works, concerning aspects such as *fault load* [Cotroneo et al., 2012], *representativeness of the faults* [Durães and Madeira, 2006; Natella and Cotroneo, 2010; Natella et al., 2012; Irrera and Vieira, 2015], *code coverage* [Van Der Kouwe et al., 2014], and *elusiveness of the faults* as well as the *relevance of their location* [Natella et al., 2010]. Nonetheless, despite all the challenges and limitations, SWIFI has been considered as the current best alternative to emulate realistic faults. It has been extensively used for multiple purposes, such as the validation of fault-tolerant mechanisms and dependability benchmarking [Durães and Madeira, 2006] in various contexts, from device drivers [Cotroneo et al., 2018] to critical [Irrera et al., 2017] and blockchain systems [Hajdu et al., 2020].

Due to its potential, over the years various techniques and tools have been proposed for SWIFI. The Generic Software Fault Injection Technique (G-SWFIT) injects realistic software faults in machine-code level using a fault library that contains the emulators for each fault type based on machine-code level and the corresponding code changes [Durães and Madeira, 2006]. The fault model and library considered in G-SWFIT have been widely accepted and incorporated in various other fault injectors. Costa et al. [Costa et al., 2003] propose a machine-code level fault injector for the Linux OS that performs all the operations on the fly (i.e., without requiring recompiling the kernel). It uses a flexible runtime kernel upgrading algorithm to allow access to system spaces and was also later extended to be compatible with G-SWFIT [Costa et al., 2009; Costa, 2013].

Ng and Chen [Ng and Chen, 1999] implemented a machine-code level fault injector that injects bugs into the kernel of a running OS to study and assess the dependability of OSs and that has since been used and updated in several works [Swift et al., 2006; Depoutovitch and Stumm, 2010; Kwon et al., 2016; Cotroneo et al., 2018]. The injected faults range from low-level (e.g., bit-flips) to high-level (e.g., memory allocation) faults [Ng and Chen, 1999]. The latter are the most relevant and intend to approximate the assembly-level manifestation of real C-level programming errors. As these faults are context-dependent, the injector disassembles the binary of a selected function in the kernel text segment and searches for proper locations in which each type of fault can be injected [Yoshimura et al., 2013]. A simplistic example of how a machine-code level fault injector operates is illustrated in *Figure 2.6*. After disassembling the original code the fault injector identifies a candidate location to inject an *off-by-one* fault, changing a `jl` (`>=`) to `jle` (`>`), which in this architecture corresponds to changing the operator in machine-code from `7C` to `7E`.



```
func:
        push   rbp
        mov    rbp, rsp
        mov    DWORD PTR [rbp-4], edi
        mov    DWORD PTR [rbp-8], esi
        mov    eax, DWORD PTR [rbp-4]
        cmp    eax, DWORD PTR [rbp-8]
        jl     L3
        mov    eax, DWORD PTR [rbp-8]
        mov    DWORD PTR [rbp-4], eax
L3:
        nop
        pop    rbp
        ret
```

```
func:
        push   rbp
        mov    rbp, rsp
        mov    DWORD PTR [rbp-4], edi
        mov    DWORD PTR [rbp-8], esi
        mov    eax, DWORD PTR [rbp-4]
        cmp    eax, DWORD PTR [rbp-8]
        jle    L3
        mov    eax, DWORD PTR [rbp-8]
        mov    DWORD PTR [rbp-4], eax
L3:
        nop
        pop    rbp
        ret
```

```
void func(int a, int b) {
  if(a >= b) {
    a = b;
  }
}
```

```
void func(int a, int b) {
  if(a > b) {
    a = b;
  }
}
```

Figure 2.6: Machine-code Fault Injector

Several tools have also been developed for the other injection levels. Natella et al. [Natella et al., 2012] assess a source-code level fault injection tool, SAFE, and compare it with binary-code level fault injection. While it allows injecting more representative faults, it does not scale well to large code-bases due to the need to recompile. Gabor et al. [Gabor et al., 2019] leverage the Clang AST Matcher to inject faults into the source-code, assessing the applicability of long-used fault models and achieving higher accuracy, as well as allowing injection in macros. Other approaches use hybrid solutions (e.g., Hybrid Software Fault Injection (HSFI) [Kouwe and Tanenbaum, 2016]), combining both source-code with machine-code level fault injection to overcome the individual limitations (e.g., lack of context of machine-code level and lack of scalability of source-code level). In short, it uses source-code level context to inject the faults, making it possible to enable or disable them using machine-code modifications without the need for rebuilding. The fact that it does not require recompiling the code for each experiment and that multiple faults can be injected at once allows for substantial savings when testing large systems (e.g., OSs). Other works also leverage recent developments (such as the rise of Clang and LLVM) and explore injecting faults at intermediate-code, which is closer to the source code and therefore easier to inject realistic faults [Lu et al., 2015].

Selecting and using a fault injector to conduct a comprehensive fault injection campaign on complex software is a difficult task, as most of them require complex installation procedures, some even including changes to the kernel that have to be conducted offline, or require specific execution modes (e.g., debug).

2.1.3 Dependability and Experimental Testbeds

Dependability research typically requires executing large sets of experiments to be representative and achieve statistical relevance. This requires devising and implementing an experimental testbed to ensure proper execution and the repeatability of the experiments.

In recent years the computational power available has increased considerably. To leverage it, several techniques have been developed at hardware level, such as *running multiple threads on a single core* [Intel, nda], *accelerating the processor for peak loads* [Intel, ndb], and *hardware virtualization* [Intel, ndc]. Such techniques can, and should, be used to expedite the experimental process, often by running multiple experiments simultaneously. This also allows reducing hardware experimentation costs to run experiments simultaneously (an issue raised by Kanoun and Spainhower [Kanoun and Spainhower, 2008]) which is a prominent concern in research. Notwithstanding, while increasing throughput is relevant, the performance and behavior of the experiments should be as identical as possible, regardless of other running experiments or tasks. Such techniques, as well as virtualization, raise some concerns, such as if, and how much, interference exists between experiments [Schwahn et al., 2019a] and how it affects the results [Novaković et al., 2013].

While the need for performance isolation is easy to understand, achieving it is far more complex. So many factors in modern computers non-deterministically influence their performance that even identifying them is not trivial. Besides differences between OSs and distributions, the architecture of the system also influences the load distribution (e.g., Central Processing Units (CPUs) shared L# caches, multiple Non-Uniform Memory Access (NUMA) nodes). To optimize resource usage modern schedulers distribute the tasks over all the cores available in the CPU depending on the current demand. Moreover, because energy efficiency is nowadays an important factor in the design of CPUs their running frequency is influenced by several factors (e.g., computational load, temperature). Besides strict performance stability, running multiple experiments should not compromise the execution of others. Concerning software isolation, virtualization has become the *go-to* solution. If corruption occurs on a given machine, it should not affect the host or other running applications. However, although it allows setting the resources that can be used by each Virtual Machine (VM), this does not guarantee performance isolation (e.g., overprovisioning) [Jing et al., 2014; Matthews et al., 2007; Gupta et al., 2006].

There are two main types of hypervisors (i.e., *Type-I* and *Type-II*), each with various solutions with respective advantages and disadvantages. *Type-I* hypervisors (also called *bare-metal*, e.g., XEN [Xen, nd], VMware ESXi [VMWare, nda]) run on the underlying hardware, interacting directly with the components (e.g., CPU, memory). As a result, they typically have better performance and security, and also offer better isolation [Matthews et al., 2007]. However, they are mostly focused on enterprise solutions (and thus usually have a higher cost), have limited hardware compatibility, and limit the machine to just using virtualization. Moreover, these are specific applications/solutions that may not meet the needs and flexibility required by researchers. On the other hand, *Type-II* hypervisors (also known as *hosted*, e.g., Oracle VirtualBox [Oracle, nd], VMWare Workstation [VMWare, ndb]) run on top of an existing OS. They are considerably more versatile, have wider hardware support, and are easier to use and to set up [IBM, nd]. Although 'traditional' *Type-II* hypervisors are considerably slower (as they emulate all the interactions with the hardware) modern hypervisors typically rely on hardware-assisted virtualization, which allows reaching near-native performance [Palit et al., 2013]. One interesting hypervisor is KVM [KVM, nd], which turns the Linux kernel into a *Type-I* hypervisor. [Hat, ndc]. Combined with QEMU [QEMU, nd] it provides all the advantages of a *Type-II* hypervisor with performances similar to those of *Type-I*.

In order to run large sets of experiments, automation is also necessary to minimize the need for interaction. This is often achieved through scripting (e.g., Bash) or programming languages (e.g., Python). However, complex automation (e.g., launching/interacting with VMs) requires more advanced solutions. *Expect* [Libes, 1995, nd] is a Tool Command Language (TCL) program that 'talks' to other interactive programs, programmatically characterizing an interaction between user-/program. Still, if one prefers/needs the functionalities/advantages of a programming language, similar implementations have also been made over the years (e.g., Java [Gavrilov, 2018], Python [Spurrier, nd]).

Another relevant concern is how to monitor a system. Most OSs provide native tools to obtain system metrics (e.g., sysstat [sysstat, nd] on Linux). Notwithstanding, they often report the data in a rather unstructured format, and various tools are necessary to monitor the most relevant resources. Concerning centralized (free/open-source) solutions, there are some well-known options such as Munin [Munin, nd] and Nagios [Nagios, nd]. However, although they are very comprehensive (their purpose is broader than just system monitoring), it is not straightforward/possible to have easy access to 'real-time' metrics. Tools similar to Netdata [Netdata, nd], which is designed to be a lightweight and highly optimized tool that provides real-time monitoring by default, are likely more suitable alternatives.

All these constraints, techniques, and configurations pose a considerable to consistent and repeatable experiments. This is further aggravated by the lack of proper and detailed guidelines on how to devise and implement an experimental testbed for dependability experiments.

2.1.4 Benchmarking

A benchmark is an instrument that allows evaluating and comparing different entities according to specific characteristics, under the same conditions, the same workload, and the same procedure [Gray, 1992]. The work on performance benchmarking started long ago and ranges from simple benchmarks that target a very specific hardware system or component to very intricate benchmarks focusing on complex systems (e.g., databases, operating systems, web servers [Vieira and Madeira, 2003]). Performance benchmarks have contributed to improving successive generations of systems, and have boosted the research on dependability benchmarking, with several works carried out by different groups following distinct approaches (e.g., experimentation, modeling, fault injection) [Koopman et al., 1997; Vieira and Madeira, 2003; Antunes and Vieira, 2010].

Benchmarking is an experimental procedure that aims at providing a practical way to measure and compare properties of systems or components, ranging from performance [Gray, 1992] to dependability and security aspects [Vieira and Madeira, 2003; Durães et al., 2004; Vieira and Madeira, 2005]. In practice, a benchmark reproduces the observations and measurements either deterministically or on a statistical basis (giving confidence in the results obtained), and allows generalizing the results to a limited extent (becoming useful beyond the particular case analyzed), which is attained by addressing the representativeness of the benchmarking process and components. The concept of benchmarking can be summarized in three words [Vieira and Madeira, 2003; Gray, 1992]:

1. **Representativeness** – include components (e.g., a dataset) that are representative of a given domain (for this work, the failure prediction domain), thus reducing the distance between the benchmarked and the real environment;
2. **Usefulness** – provide a useful representation of the entities under analysis, capturing the essential elements of the domain and characterizing their features, thus allowing one to use the results for choosing the best alternative or to guide improvement;
3. **Agreement** – provide a standard procedure to assess relevant metrics related to an entity on which users can agree, allowing measurement results to be accepted.

In order to analyze the different approaches for OFP, it becomes relevant to have a way to fairly compare and assess their performance. Hence, a benchmark is likely the most logical approach. A benchmark tool has different components, mainly [Gray, 1992]:

- **Metrics** – are used to assess the entities under observation. The metrics are one of the most important components to be defined, as it is based on them that the alternatives will be compared. They should be computed during the benchmark execution, and must be understood in relative terms, not as absolute measures;
- **Workload** – represents the set of operations that will be executed by the system while the benchmark is running. They are usually built to exercise the specific system under benchmarking, and although there have been multiple works of their generation, it still remains a challenging task that can influence the relevance of the benchmark;

- **Benchmarking Procedure** – defines the setup and procedure required to execute the benchmark and the various steps to be performed throughout its execution.

Performance is no longer the only important factor when selecting alternative systems or components, and dependability is playing a more and more determinant role. Dependability benchmarks can be defined as a standard procedure to assess the dependability-related measures of a given system in the presence of faults [Vieira and Madeira, 2003; Crouzet and Kanoun, 2012]. Compared to performance benchmarks, they have additional components, such as *the faultload* (a set of faults and stressful conditions) and *dependability measures* (measures that characterize the dependability of the system under benchmark) [Vieira and Madeira, 2003]. Dependability measures are associated with the various attributes that comprise *dependability* and have been defined as [Crouzet and Kanoun, 2012; Avizienis et al., 2004]: *reliability* measures the time to failure, *availability* measures the fraction of time the system delivers correct service, *maintainability* measures the time to service restoration (since the last failure), and *safety* measures the time to catastrophic failures. Several dependability benchmarks have been proposed in recent years (e.g., [Crouzet and Kanoun, 2012; Cotroneo et al., 2017]).

A benchmark for failure prediction must address specific properties for the results to be sound, and to minimize inaccuracies due to the measurement procedure, namely [Gray, 1992; Vieira and Madeira, 2003]:

1. **Ease of installation and use** – the benchmark should be composed of a program ready to use or a document specifying how to implement the benchmark, the tools needed, etc. The user should be able to analyze failure prediction models with minimum effort;
2. **Promptness** – the benchmark execution should take the shortest time possible. Promptness increases the usability of the benchmark and of the failure prediction models, and potentially reduces the cost that one has to allocate for the benchmarking task;
3. **Non-intrusiveness** – the benchmark must require minimal or no changes in the entities under analysis, which in this context are the predictive models;
4. **Portability** – the benchmark must allow comparing alternative failure prediction models, which can be based on diverse approaches. Considering different application scenarios and target systems should be allowed;
5. **Repeatability** – different executions of the benchmark must lead to the same results on a deterministic basis or in statistical terms. The results should not depend on a single execution of the benchmark;
6. **Representativeness** – the results from the benchmark must be representative of real scenarios, i.e., the prediction models must behave similarly (in relative terms) when working on the target system in a real situation.

Another concept that has fairly recently been proposed is that of scenarios [Antunes and Vieira, 2015], which applied to OFP, are simply realistic situations of failure prediction that depend on the criticality of the target system. As the usefulness of a benchmark is highly dependent on the metrics used to assess the entities, it is also dependent on the adequacy of those metrics for a specific scenario [Antunes and Vieira, 2015]. Scenarios should be based on the technical needs and business impact of the systems in an organization (e.g., business-critical), by means of requirements in terms of the level of dependability that should be satisfied and the cost of mitigating the predicted failures before their occurrence.

2.2 Machine Learning (ML)

Although there is no formal consensus on what *intelligence* really is, it can be informally defined as the general ability of an agent to perceive, understand, predict, and manipulate a world far more complicated than itself to achieve goals in a vast range of environments [Legg and Hutter, 2006; Russell and Norvig, 2021]. Artificial Intelligence (AI) is concerned with 'building intelligent entities, machines that can compute how to act effectively and safely in a wide variety of novel situations' [Russell and Norvig, 2021]. Artificial Intelligence (AI) has already been around for some time, and in conjunction with the previous definition, it can be explained as the science of building intelligent machines, including intelligent computer programs [McCarthy, 2007]. From that evolved the idea of creating systems that can adapt and learn within an environment, which came to be known as Machine Learning (ML). Briefly, ML is a subfield of Computer Science (CS) that gives computers the ability to learn without being explicitly programmed [Samuel, 1959]. Despite its recent intensive development, it has also been around for quite some time. Depending on the type of feedback available to the learning system, ML approaches can be classified into *supervised learning* (uses a set of labeled data to learn a function that maps the data into the appropriate label), *unsupervised learning* (tries to find the hidden structure of a set of unlabeled data), and *reinforcement learning* (learns an optimal policy to solve a sequential decision making problem, using rewards received from interactions with an environment) [Ayodele, 2010; Duboue, 2020; Marsland, 2014].

The work conducted in this thesis relies almost entirely on supervised learning. Thus, notwithstanding the importance of all the remaining types of learning, this section will focus mostly on supervised concepts and techniques.

Different types of algorithms can be used to learn a computational model to solve a given problem. However, to create an accurate model it is crucial to thoroughly prepare the data, as it will directly influence its performance. Also essential to the analysis of ML algorithms is the ability to compare them and be able to state which are best at what.

A generic ML approach, based on and adapted from [Duboue, 2020], can be seen in *Figure 2.7*. Duboue [Duboue, 2020] explicitly depicts the relevance of data analysis in the whole process. The insights it provides allow for more informed decisions, such as selecting the target ML algorithms and the evaluation metrics. Although the various steps will be discussed in the next sections, another important consideration is the split of the data into different subsets, *train* and *test*. The train data will be used to train the models and optimize their hyperparameters, also considering the impact of different ML techniques (e.g., *data sampling, feature selection/extraction*). The test set will ultimately be used to estimate the performance of the best models.

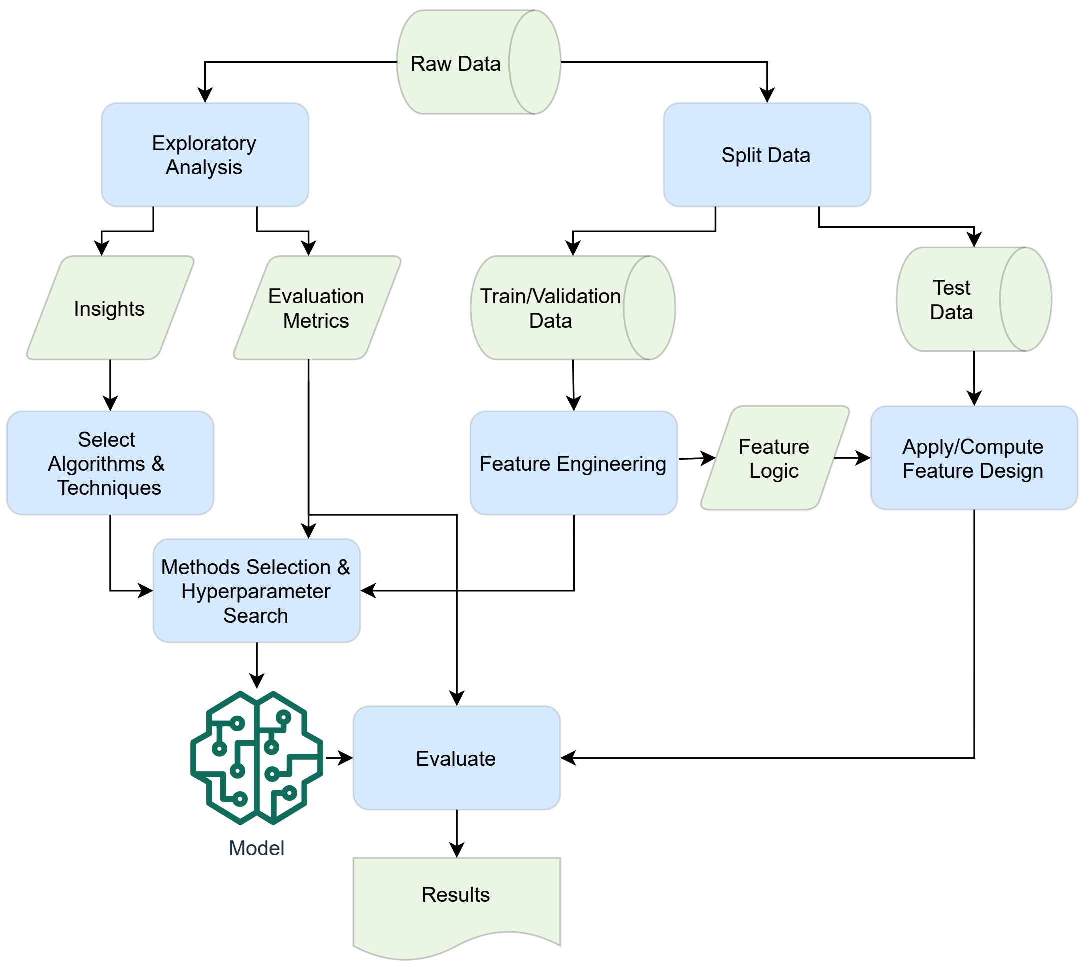


Figure 2.7: Generic ML Pipeline, adapted from [Duboue, 2020]

2.2.1 Data Analysis

To use ML it is necessary to explore and thoroughly identify characteristics to transform raw data into working information. This will provide details and the underlying characteristics of the problem.

Several techniques are available, which in general can be grouped into two approaches. Because data are organized in distributions, with *descriptive statistics* it is possible to analyze them through some statistics that describe or summarize them, taking into account its characteristics, such as centrality (*mean, median*, and *mode*) and spread (*standard deviation* and *variance*). *Exploratory Data Analysis (EDA)* has its focus on the data, mainly using graphical tools that provide a different perspective. Through EDA it is possible to visualize the centrality and spread of the data, using *boxplots*, *scatterplots*, *histograms*, and *runcharts*, amongst others. These graphical tools allow the user to perceive patterns, distributions, outliers, and relationships between features that would otherwise not be easily detected or understood with other analyses.

2.2.2 Data Preparation

Due to the fact that nowadays most data gathering processes are loosely controlled, many of the datasets usually have missing, noisy or unreliable, redundant or invalid data. Moreover, if these data are not carefully prepared the model may not be representative of the situation in study. Because of that, data preprocessing is one the most important steps in ML, usually consuming a considerable amount of development time. Preparing the data includes several techniques, namely *cleaning*, *transformation*, *feature reduction* and *extraction* [Kotsiantis et al., 2006; Duboue, 2020].

In traditional ML the basic requirements of the data for its models are fairly simple. It needs a dataset of examples of the scenario it represents as large as possible/needed, containing enough detail to describe it and its outcome (for supervised learning). It is a simple table where the columns are divided into a set of descriptive features and a target, and each row represents an instance, that contains a value for each feature and target [Kelleher et al., 2015]. Notwithstanding, some of the most recent advances on ML use other sources of data, such as images, signals, text, and speech [Duboue, 2020].

In order to prepare the data, some steps should be taken to maximize the performance of the algorithms. *Data cleaning* techniques include *noise reduction* and *outliers detection* [Aggarwal, 2013], *handling missing values* [Bruha and Franek, 1996; Grzymala-Busse and Hu, 2001; Lakshminarayan et al., 1999], and analyzing the data for *data inconsistency.*

Data transformation techniques include *discretization, aggregation/decomposition*, and *normalization.* Normalization is one of the most common techniques, and it is the scaling of feature values so that they fall under a certain range or relate to the feature values distribution. The most common methods to achieve this are *min-max scaling* (the data is scaled to a fixed range, usually 0 to 1) and *z-score normalization* (also known as *standardization*), where all features will be rescaled so that they will have the properties of a standard normal distribution, with zero mean and a standard deviation of one.

The complexity of any model usually depends on the number of inputs it has, as it influences both the time and space complexity to train it. In order to facilitate the development of a model, it may be important to reduce its dimensionality. Due to the way most of the information is gathered nowadays, without a purpose and controlled supervision, most of the datasets have irrelevant or redundant features in them. The removal of these features is called *feature selection.* There are a few methods to perform feature selection, although they are usually grouped in three: *filter*, *wrapper*, and *embedded.* The other approach to reduce dimensionality is *feature extraction.* The goal is to find a new set of dimensions that are a combination of the original ones. These methods can be supervised or unsupervised and some of the most widely used are Principal Component Analysis (PCA) [Silipo et al., 2014] and Linear Discriminant Analysis (LDA) [James et al., 2013]. In recent years deep learning algorithms (e.g., Convolutional Neural Network (CNN)) have also been shown to be very effective at extracting features from complex data.

Instance selection methods are used in order to help the algorithms to cope with the infeasibility of very large datasets. It becomes an optimization problem to minimize the data size while keeping its quality and representativeness. Instance selection can be achieved through *sampling*, *boosting*, *prototype selection*, and *active learning* [Ghosh, 2004]. One of the major means of instance selection is sampling (e.g., stratified sampling), whereby a sample is selected for testing and analysis, and randomness is a key element in the process [Liu and Motoda, 2013]. Ultimately, a sophisticated procedure with a subset of the data can outperform a less sophisticated one using the whole dataset [Friedman, 1997].

Imbalanced datasets may also lead to models that are not able to generalize (i.e., *overfitting*). *Undersampling* techniques seek to reduce the number of samples of the majority class in the dataset [Duboue, 2020]. As a result, the overall number of records in the dataset is decreased, also shortening training time. Whilst this can lead to better performance on the underrepresented classes, it may also lose some valuable information with the discarded items. Over time, several techniques have been proposed such as Random Undersampling [Liu, 2004] and Instance Hardness Threshold [Smith et al., 2014]. *Oversampling*, on the other hand, seeks to increase the number of samples from the minority class. The obvious advantage is that it does not lose information since all the samples are kept. It does, however, increase the size of the dataset and consequently the time to train the model. Some of the methods to achieve this are Random Oversampling [Liu, 2004] and Synthetic Minority Over-sampling Technique (SMOTE) [Chawla et al., 2002].

2.2.3 Learning Types and Algorithms

The type of feedback available in the data determines the types of learning that can be used, as illustrated in *Figure 2.8*. Three main types of learning exist: *supervised learning, unsupervised learning,* and *reinforcement learning* [Russell and Norvig, 2021]. Due to the scope of the work conducted in this thesis, supervised learning will be described in more detail.

2.2.3.1 Supervised

Supervised learning is by far the most common type of ML. This type of learning contains the data and the expected output for every combination of inputs. The goal is to learn a mapping between the input variables and their outputs in such a way that for a new set of inputs it will be able to predict the output. Based on the type of problem it can be either *classification* (predicting a categorical value) or *regression* (predicting a continuous value). Most of the algorithms described in this section can be used for both classification and regression problems.

Some of the most common supervised algorithms are Naive Bayes, Logistic Regression, Linear Regression, Decision Tree (DT), Support Vector Machine (SVM), Neural Network (NN), Convolutional Neural Network (CNN), and Ensemble

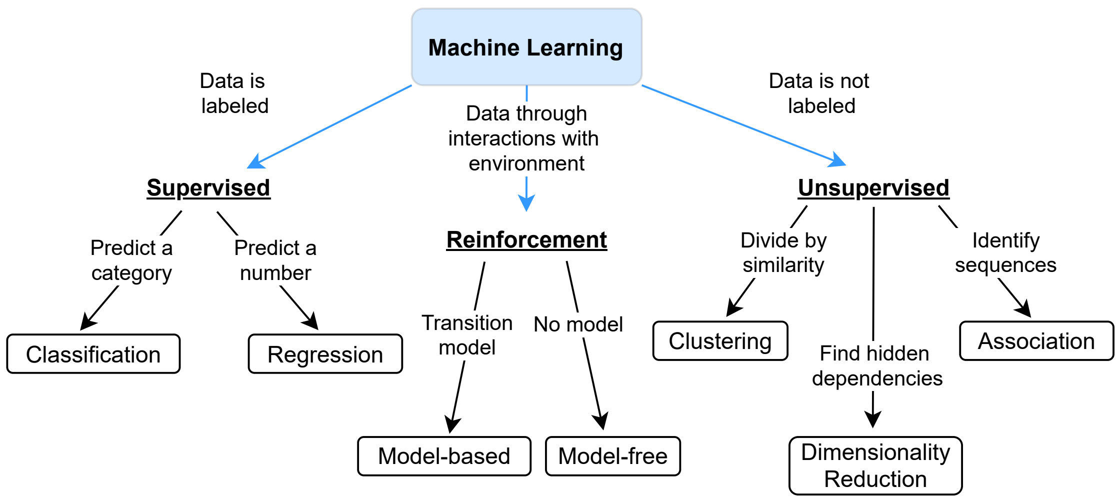


Figure 2.8: Machine Learning Types

Methods. The most relevant for the work conducted on this thesis will be briefly detailed further.

Decision Tree (DT)
Top-down, recursive, and one of the most intuitive prediction models. They construct models based on attributes present in the training data (guided by a criterion, such as *entropy* or *information gain*) using a decision tree as a predictive model. It makes predictions of the value associated with an instance by traveling from a root node to a leaf [Shalev-Shwartz and Ben-David, 2014]. If the target variable is continuous the models are called *regression trees* and *classification trees* otherwise.

Due to the computational cost of trying to cover the whole solution space to find the tree that optimizes the problem, practical decision tree learning algorithms are based on heuristics. Decision tree algorithms usually suffer from generating very large trees, creating low empirical risk but having a high true risk. A solution to avoid that is to limit the number of iterations, creating a bounded tree, or prune it, reducing the size but keeping the empirical error [Shalev-Shwartz and Ben-David, 2014]. Some of the most well-known tree predictors are Iterative Dichotomizer 3 (ID3), C4.5, and Classification and Regression Trees (CART).

Support Vector Machine (SVM)
Classification algorithms that are able to predict both linear and non-linear data in high dimensional feature spaces. This high dimensionality raises both computation and sample complexity challenges. To face the sample complexity the Support Vector Machine (SVM) searches for 'large margin' separators, by maximizing the distance between the samples of each class and the hyperplane that separates them [Shalev-Shwartz and Ben-David, 2014].

If the data are not linearly separable it is also possible to map it to a high dimensional space, where it is easier to classify with linear decision surfaces. Mapping all the data to the high dimensional space may be too expensive, so it is possible to use *kernels*, in order to avoid explicitly making the mappings. Some of the most used kernels are the *polynomial* and the *gaussian*.

Neural Networks (NNs)
A model of computation loosely inspired by the structure of neural networks in the brain. NNs are universal function approximators, which means that they can approximate any continuous function (to an arbitrary degree of accuracy) [Russell and Norvig, 2021]. It can be described as a directed graph, where the nodes are the neurons and the edges are the links between them. Each neuron has as input a weighted sum of the outputs of the neurons connected to its incoming edges [Shalev-Shwartz and Ben-David, 2014]. *Deep learning* refers to multi-layered networks and is a subject that has been gaining relevance in recent years. Each layer transforms the representation of the previous layers into deeper and more abstract levels [Hosseini et al., 2020]. This process inherently allows and works as feature selection/extraction.

For an NN with N number of layers, the layers $H_0, ..., H_{N-1}$ are usually called *hidden layers*, H_0 the input, and H_N is the output layer. In general, the architecture of a NN is comprised of a *topology* (e.g., feedforward, recurrent), an *activation function* (e.g., sigmoid, Rectified Linear Units (ReLU)), a *learning algorithm* (e.g., gradient descent, backpropagation), and *layers* (e.g., convolutional, pooling). In recent years several deep learning architectures have been proposed but most can be classified into high-level categories, such as CNNs and Recurrent Neural Networks (RNNs) [Heaton, 2015; Hosseini et al., 2020].

Deep Neural Networks (DNNs) have achieved remarkable state-of-the-art results in some fields, such as computer vision, speech recognition, and natural language processing [Witten et al., 2016]. DNN can achieve good results when there is spatial or sequential relation (e.g., adjacent pixels in an image, or words in a sentence). While there have been some recent works on using DNN on tabular data (e.g., [Arık and Pfister, 2020]) it has not yet been able to achieve as relevant results. Additionally, DNNs are *black-box* models, as the combination of multiple complex hidden layers means that it is hard/complex to properly interpret the decisions of the model.

Ensemble Methods
Ensemble methods are a compilation of several independent algorithms whose predictions are gathered to work as a whole. They contain a number of learners called *base learners* (or *weak learners*). Ensembles can be either *homogeneous* (i.e., use a single base learning algorithm) or *heterogeneous* (i.e., use multiple base learning algorithms) [Zhou, 2012]. Some of the state-of-the-art algorithms currently available are ensembles that use the Decision Tree (DT) as the base learner, such as Random Forest (RF), based on the *bagging* paradigm, and Gradient Boosting (and the specific implementation of eXtreme Gradient Boosting (XGBoost)), based on the *boosting* paradigm. Notwithstanding, heterogeneous ensembles can explore the different bias and diversity of each algorithm and often lead to better results [Bian and Wang, 2007; Costa et al., 2018].

A good ensemble is made of base learners as diverse as possible and their outputs should be combined in such a way that correct decisions are amplified and incorrect ones are canceled out [Polikar, 2006]. However, measuring diversity (i.e., the difference between the individual learners) is complex, as there is no consensus or formal definition [Sesmero et al., 2015]. Several metrics have already been proposed (e.g., Q Statistic, disagreement measure), although the correlation between diversity and performance is not fully understood [Kuncheva and Whitaker, 2003]. The combination method also plays a crucial role. For classification tasks the most common techniques are based on voting: *i) plurality voting*: each classifier votes for one class label and the final prediction is the class with more votes; *ii) majority voting*: the final prediction is the class that obtains more than half of the votes (a rejection option will be given if there is no majority); *iii) soft voting*: for algorithms that provide class probability outputs it averages the predictions for each class and selects the one with the highest probability (thus taking into consideration both the prediction and the confidence of the model in the decision) [Zhou, 2012]. Another prominent approach is *stacked generalization*, which uses the concept of a *meta-learner* trained using the outputs of the base learners as the input features.

2.2.3.2 Unsupervised

Unsupervised learning has no supervision in the learning process, that is, for a combination of inputs, there is no class or label for it. In this type of learning the aim is to find a structure or distribution in the input space so that recurring patterns can be found. The most common approaches are *clustering*, *dimensionality reduction*, and *association*. Clustering is by far the most common, and it tries to find structures that exist in the data, grouping the samples into groups of maximum commonality, by maximizing the intra-class similarity while minimizing inter-class similarity [Jiawei Han, 2011]. Some of the most common algorithms are Density-Based Spatial Clustering of Applications With Noise (DBSCAN) (a density-based clustering algorithm that tries to identify and distinguish points in clusters from noise, based on its density), Hierarchical Clustering (HC) (uses linkage rules to produce a hierarchical sequence of clustering solutions), and k-Means (a centroid adjustment algorithm). In recent years there have also been some interesting developments on *generative models* (an unsupervised learning task that tries to learn the statistical distribution of the data) using deep learning, such as Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs) [Hosseini et al., 2020].

Although unsupervised learning is useful for many real-world problems (which are very often unlabeled), the work conducted in this thesis focuses mostly on supervised learning.

2.2.3.3 Reinforcement Learning

Reinforcement learning algorithms create models that map situations to actions through a series of reinforcements (*rewards* and *punishments*) obtained from interactions with the environment. The model learns through those interactions and observing the results they produce. It is not told explicitly which actions to take, it must discover which actions yield the most reward by trying them, affecting not only the immediate reward but all the subsequent rewards. Trial-and-error and delayed rewards are the two characteristics that distinguish this type of learning from the remaining [Marsland, 2014]. Reinforcement learning has also been combined with deep learning, leading to a field called *deep reinforcement learning*. In recent years reinforcement learning has been very successfully used for various problems, such as autonomous driving [Kiran et al., 2021], trading and finance [Gao, 2018], Natural Language Processing (NLP) [Choi et al., 2017], and games [Holcomb et al., 2018].

Over the years several different algorithms have been proposed, which can be categorized into *model-based* and *model-free* reinforcement learning [Russell and Norvig, 2021]. Briefly, model-based approaches use a transition model of the environment to help interpret the reinforcements and make decisions. On the other hand, model-free approaches neither know nor learn a model of the environment. They learn a more direct representation of how to behave through *action-utility learning* (learning a quality function) or *policy search* (learning a policy that maps directly from states to actions) [Russell and Norvig, 2021].

2.2.4 Evaluation

A learning model is good if it produces correct predictions on unseen examples. To do so, its performance is usually measured by its error/success rate. Albeit some of the problems have undifferentiated costs for different errors, that is, the cost of misclassifying A as B is the same as misclassifying it as C, that is not the case for some more complex problems. It may be the case that wrong decisions are not equally costly, requiring the implementation of a more complex error loss function. Despite the fact that the errors are one of the main criteria for evaluating an algorithm, it should be kept in mind that there are others (e.g., computational time, interpretability), some of them being dependent on the problem at hand [Turney, 2002].

Regardless of all the factors that can be taken into account when evaluating an algorithm, one should not forget that whatever conclusions that may arise are conditioned by the dataset with which the model was developed. Also, the comparison that can be made using the results is not domain-independent, because we are not comparing the expected error rates of a learning algorithm in general, but rather for a specific application and only as long as the sample used represents the target application. When it is said that a classification algorithm is good, it is only a qualification of how well its inductive bias matches the properties of the data [Alpaydin, 2014]. In fact, there is no universal algorithm that is on average better than random search for every situation, as stated by the No Free Lunch Theorem [Wolpert, 1994].

2.2.4.1 Training and Testing

A fundamental problem in ML is how to obtain a realistic estimate of the prediction error of a model. This task is of particular relevance when the dataset is not large and the underlying distribution is not known [Borra and Di Ciaccio, 2010]. This estimate is important as it is based on its value that a model will be chosen instead of others due to having a better prediction performance.

ML usually works with two main sets of data: *training*, and *test*. The training set contains the data that are going to be used to fit the model while it is in training. Within the training set, a *validation* subset may also be created to evaluate the candidate models and control overfitting. The test set estimates the generalization error of the final model. In theory, the test set should only be used once to avoid data leakage and for each evaluation a new set of data should be used. Notwithstanding, this is not feasible for many (if not most) real problems, and thus it should be taken into consideration that by reusing the test data the results will overfit to that set [Duboue, 2020]. The division between these sets of data is not trivial, as when the model is in training it will not have access to the test set, hence the selection of the subsets must be representative of the problem. To create them there are some options [de Sá, 2012], such as *resubstitution, holdout, partition/leave-one-out*, and *bootstrap* methods. One of the most used techniques is a partition method, *cross-validation*. It can be used to improve the prediction. Briefly, after the data is split into k disjoint subsets the model is then trained k times. Each time one of the subsets is left out and then used to compute the prediction error. Additionally, *stratification* can be used to ensure that the representation of the classes is similar in all the resulting folds. The final performance of the model is the average of the results of all the folds.

With every model created using a dataset a problem arises that is known as the *bias/variance dilemma*. Bias is a source of error derived from wrong assumptions made by the algorithm, while variance is a source of error due to the sensitivity to changes in the training set. When a model cannot fit the data it is known as *underfitting*, and will usually have low variance and high bias. However, if the bias is kept low the model may fit the data too well (noise and random events included) and have a high variance, which is known as *overfitting*. Thus, there is the need for a trade-off between minimizing the bias and the variance: a choice between creating more complex models (low-bias hypotheses that fit the training data well) and simpler models (low-variance hypotheses that generalize well) [Russell and Norvig, 2021]. While there is no silver bullet to find the optimal trade-off, Ng [Ng, 2017] proposes a systematic approach. To address high bias the following techniques can be used: *train longer, train a more complex model, obtain more features, decrease regularization,* and ultimately *consider a new model architecture*. On the other hand, to address high variance: *obtain more data, decrease the number of features, increase regularization,* and (once again) *consider a new model architecture*. Finally, some techniques (e.g., ensemble learning, resampling) can be used to manage bias and variance, and continuously assessing the performance of the models (e.g., through a validation set) can be used to control underfitting/overfitting.

2.2.4.2 Performance Metrics

In order to evaluate an algorithm, its performance must be measured. Several performance metrics exist and are dependent on the type of task (e.g., classification, regression). However, which metrics to use should be carefully selected, as they are not independent of the data, and thus can be influenced by their distribution [Sokolova and Lapalme, 2009]. If this is not carefully considered, it may lead to misleading conclusions.

Several metrics have been proposed for *classification* tasks. Some of the most common and relevant are the *confusion-matrix, error rate, accuracy, precision, eecall, F_β-Score, informedness, markedness, sensitivity, specificity,* and *Receiver Operating Characteristics (ROC). Regression* tasks predict continuous values, thus metrics to assess their performance require a different approach. Some of the most common are *Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Square Error (RMSE)*, and *coefficient of determination.*

On the other hand, evaluating the performance of an algorithm for a clustering problem is not as straightforward as for classification or regression. Clustering metrics can be divided into *internal validation* (e.g., *Sum of Squared Errors (SSE) and silhouette coefficient*) and *external validation* (e.g., *rand index, mutual information, homogeneity, completeness*, and *v-measure*) [Tan et al., 2005].

2.2.4.3 Assessing and Comparing the Performance of Algorithms

After implementing and compiling the results of the target algorithms there is a need to compare them. For that, there are several tests, depending on the characteristics of the data (e.g., number of categories, parametric, dependency). Two hypotheses are postulated: H_0 (*null hypothesis*), the samples come from the same population; and H_1 (*alternative hypothesis*), the samples do not come from the same population. Significance tests ensure whether or not the differences are statistically significant, with some value of confidence [Juristo and Moreno, 2013].

Due to the representation available from the sample data, the conclusion about accepting or rejecting H_0 may be wrong. Those errors can be divided into two types (which can be seen in *Table 2.1*): *type II* error is not rejecting H_0 when it should (it is false), and *type I* error is rejecting H_0 when it should not, which means wrongfully rejecting a previous idea of truth [Field, 2013].

Table 2.1: Error Types

		Reality	
		$H_0 = true$	$H_0 = false$
Decision	H_0 is not rejected	OK	Type II Error
	H_0 is rejected	Type I Error	OK

When there are multiple algorithms that need to be compared it is necessary to use specific tests to do so. However, as multiple hypothesis testing requires several comparisons the probability of getting a significant result by chance adds up. To deal with that some *corrections* must be applied in order to adjust the confidence level so that the probability of observing significant results remains below the defined level, such as the Bonferroni correction or Holm procedures [Field, 2013].

It is worth noticing that when comparing different classification algorithms we are only testing whether they have the same expected error rate. If they do, it does not mean that they made the same mistakes. Additionally, the standard applications of these tests usually consider that all the misclassifications have the same cost. If it is not the case the tests need to take that into account.

2.2.5 Time Series

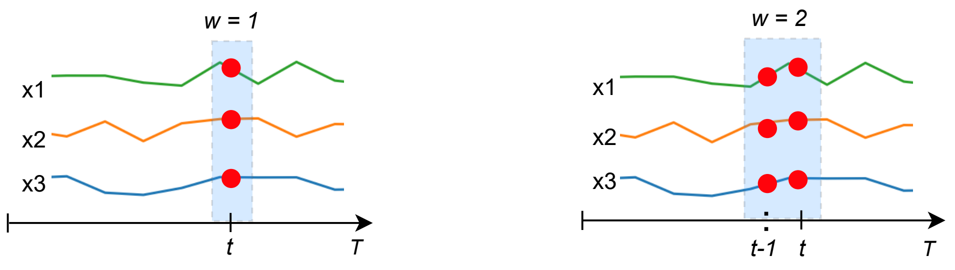
A time series is a collection of observations made sequentially in time. They exist and can be studied in many scientific fields, such as meteorology and finance. There are multiple reasons why the study of time series is important: *i)* the ability to predict the future based on the past; *ii)* to control and understand the process generating the time series; *iii)* the description of the prominent features in the series [Bontempi et al., 2013]. Time series data and sequential data are very similar, and although sequential data (e.g., gene sequences) does not have reference to time, the order of the data is important. Thus, many of the approaches used for time series can also be used for sequential data [Amr, 2012].

The first approaches for forecasting time series data (i.e., a prediction problem in the context of time series) were statistical models, such as AutoRegressive Integrated Moving Average (ARIMA) and Generalized AutoRegressive Conditional Heteroskedasticity (GARCH) [Bontempi et al., 2013]. Although they perform well with simple problems, due to their statistical nature they make assumptions about the data and are not able to detect or extract knowledge from more complex datasets, which means they are not useful for many real applications [De Gooijer and Hyndman, 2006]. A typical and important characteristic of time series is that the sequence of the values is important, there is a dependence between observations. This is called autocorrelation and it is a way of measuring and explaining the internal associations between observations in a time series.

The prediction can be either *one-step* or *multi-step* (far more complicated due to the accumulation of errors, reduced accuracy, and increased uncertainty [Tiao and Tsay, 1994]) depending on the prediction horizon. Additionally, it can be either *univariate* (only one independent variable) or *multivariate* (multiple independent variables, increased complexity) [Santos and Kern, 2016]. Time series data can also be divided into *stationary* and *non-stationary*. Being stationary means that there is no systematic change in mean and variance [Chatfield, 2016]. Conversely, non-stationary is defined by their varying statistical properties with time. Statistical models have tried to adapt to cover these data, yet, due to their strong assumptions they still present limitations. On the other hand, some ML algorithms can model the data with limited assumptions (e.g., NN, which have been described as building the model based on the data without the necessity of its previous specification [Zhang, 2003]) and may therefore be useful in such scenarios.

Due to the amount of time series data nowadays (e.g., small scale devices and every type of sensor) and the necessity to explore them, ML algorithms have been used to improve time series analysis and have established themselves as good (in some cases, better) alternatives to classical statistical models [Ahmed et al., 2010; Palit and Popovic, 2006]. Although many algorithms are not able to directly handle time series data, with some techniques they can be converted into a standard supervised problem, which will then have all the plethora of supervised ML algorithms available. Still, most algorithms will ignore the autocorrelation structure [Vilar, 2009]. Other attributes, such as high dimensionality, correlation, and noise, common in time series data also raise some challenges on the use of ML algorithms.

One of the techniques to process time series data so that they can be viewed as a supervised learning problem is to use previous time steps as features. This method is called the *sliding window* (*lag* or *lag method* in statistics) and is exemplified in *Figure 2.9*. The number of previous time steps to be considered, w, is called the *window width* or *size of the lag*. Although this technique performs well, it does not consider correlations between the predicted values [Dietterich, 2002]. Other approaches (e.g., Recurrent Sliding Window, Hidden Markov Models) provide different alternatives to consider past data [Dietterich, 2002].



a) Window Size: $w = 1$ b) Window Size: $w = 2$

Figure 2.9: Sliding Windows-Based Time Series Analysis

The interdependence between time series instants (i.e., $t+1$ is related to t, *autocorrelation* [Bisgaard and Kulahci, 2011]) may also raise some issues when splitting the data to estimate the performance of the model, as it violates the *i.i.d.* (i.e., independent and identically distributed) assumption. Even within the ML field, this remains somewhat of an open issue, as there are multiple approaches available but none is ideal (i.e., they all have advantages and disadvantages). The main issue relies on the premise that 'future' data should not be used to make predictions about the 'past'. As overviewed by Bergmeir and Benítez [Bergmeir and Benítez, 2012], to address this, several works reserve data at the end of the series for prediction, to create a clear distinction between the 'past' (training) and the 'future' (test) (often called *out-of-sample* [Tashman, 2000]). However, this approach has certain limitations, such as not using all the data and the fact that this allows only for one forecast per series, a horizon that may not be representative. While this may be mitigated by using multiple test periods (e.g., *rolling-origin*) part of the issue (i.e., not using all the data) still remains. Although this premise can be easily understood, some works (e.g., [Bergmeir et al., 2018]) contest its relevance and argue that, in most uses of ML algorithms with time series, classic cross-validation (which will mix both 'past' and 'future' data) can be used to create accurate models while minimizing overfitting.

2.2.6 Adversarial Machine Learning (ML)

In recent years several works have shown that many (if not most) state-of-the-art ML models are very sensitive to small/imperceptible variations in the data [Szegedy et al., 2014; Goodfellow et al., 2015; Chen et al., 2019a]. Such data samples became commonly known as *adversarial samples* [Goodfellow et al., 2015; Biggio and Roli, 2018]. A simple example documented by Goodfellow et al. [Goodfellow et al., 2015] is shown in *Figure 2.10*, where an image of a panda with some imperceptible (but crafted) noise is predicted as a gibbon with high certainty.

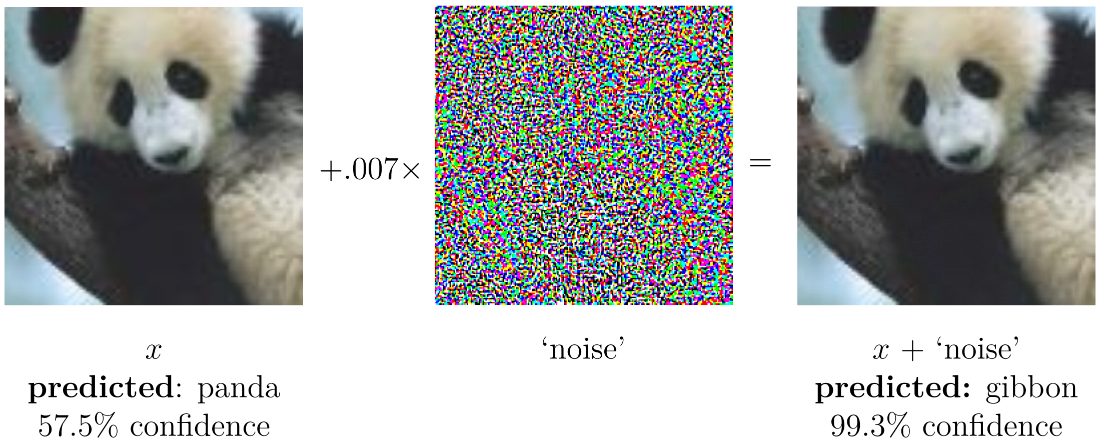


Figure 2.10: Adversarial Example [Goodfellow et al., 2015]

Adversarial environments have been extensively studied in several domains, from image classification to speech recognition (e.g., [Szegedy et al., 2014; Carlini and Wagner, 2018]), with a specific focus on NN-based algorithms. In fact, it has been demonstrated that state-of-the-art image classification DNNs were fooled by minor imperceptible changes (e.g., [Szegedy et al., 2014]) or carefully positioned stickers (e.g., [Eykholt et al., 2018]). It has also been observed that other well-known algorithms such as SVM and DT are also susceptible to adversarial samples and that adversarial samples can often be transferred and compromise models using other architectures of even algorithms (e.g., [Wu et al., 2018; Papernot et al., 2016]). Adversarial samples are typically studied from the perspective of an attacker (i.e., intending to exploit/fool the system). However, they can also be used to assess, and subsequently improve, the robustness of ML models to variations in the data (such as those that will inevitably occur in production systems). Extensive research has been done in recent years on adversarial environments, including various surveys [Chakraborty et al., 2018].

For security purposes, it is logical to assume that the attacker will try to maximize the confidence of the classifier on the desired output class. Still, a more common goal is to find the minimal perturbation that leads to misclassification [Goodfellow et al., 2015]. In general terms, adversarial samples are obtained by minimizing the distance of the adversarial sample x' to the corresponding source sample x under the constraint that the predicted label is different (i.e., $\min_{x'} d(x, x')$ s.t. $f(x) \neq f(x')$) [Biggio and Roli, 2018]. The difference between the perturbed sample and the original is typically measured using L_p-norms [Sharif et al., 2018]. Simplifying, L_0 measures how many features were perturbed, L_1 measures the difference using the Manhattan distance, L_2 uses the Euclidean distance, and L_∞ measures the largest (feature) variation. Adversarial samples are crafted by minimizing one of these norms. Notwithstanding, some works argue that L_p-norms are both unnecessary and insufficient and thus this still remains an open issue [Sharif et al., 2018].

There are two types of errors that can be leveraged [Biggio and Roli, 2018]: *error-generic* (i.e., misleading classification regardless of the output class) and *error-specific* (i.e., misclassifying as a specific output class) evasion attacks. While these two goals have been significantly researched and can be solved through a gradient-based attack for differential algorithms (e.g., NN, SVM with differentiable kernels), non-differentiable algorithms (e.g., DT, RF) require considerably more complex solutions (and do not have such a relevant related body of work). Additionally, different levels of the attacker's knowledge can also be considered [Biggio and Roli, 2018]:

- **white-box** – the attacker knows everything about the system/model (e.g., training data, learning algorithm, model hyperparameters) and can conduct a worst-case evaluation;
- **gray-box** – the attacker has limited knowledge of the system (although various scenarios can be devised, such as feature representation and/or learning algorithm) but not the training data and model hyperparameters;
- **black-box** – the attacker has no knowledge of the system (such as the feature space) and has only the feedback given through labels/confidence and the idea of the task of the system and input features/representation (i.e., it is partial, but not totally absent).

Several solutions have been developed over the years to create adversarial samples within the various knowledge levels. More recently, some approaches have also used generative models to create adversarial samples (e.g., GANs [Zhao et al., 2018; Song et al., 2018] and VAEs [Ren et al., 2020]).

To improve the robustness of the models against adversarial samples, several techniques (also known as defenses) have been developed [Biggio and Roli, 2018]. A 'simple' heuristic-based approach, *adversarial training*, relies on including adversarial samples in the training data [Kantchelian et al., 2016; Goodfellow et al., 2015]. *Data augmentation* (i.e., augmenting the training data with *hard positives*, samples that are not classified correctly with high confidence) has also been used to improve stability to input changes [Zheng et al., 2016; Kuznetsova et al., 2015; Zhao et al., 2020]. Additionally, several works are based on optimization approaches, thus providing formal robustness guarantees (e.g., game theory [Globerson and Roweis, 2006]; formulating adversarial learning as a minimax problem [Xu et al., 2009]). Another type of defense is based on detecting and rejecting samples that are too far from training data (e.g., [Meng and Chen, 2017; Wild et al., 2016]). Notwithstanding, several works have shown that the effectiveness of these defenses is limited to the scenarios considered and fall short to different or novel attacks. Ultimately, there is no single solution that can be used to improve the robustness of the models, and this is still an open issue.

2.3 Related Work

This section presents and discusses the most relevant related work on fault injection and experimental testbeds, benchmarking, ML, and OFP.

2.3.1 Fault Injection and Dependability Testbeds

Over the years fault injection has been used to assess and improve the dependability of multiple systems, such as device drivers [Cotroneo et al., 2018], libraries [Bhat et al., 2021], virtual machines [Cerveira et al., 2017], cloud platforms [Cotroneo et al., 2019, 2020], and embedded [Jeong et al., 2017], critical [Irrera et al., 2017], and blockchain systems [Hajdu et al., 2020]. However, due to its inherent challenges, fault injection targeting large and complex systems is not so common. Durães and Madeira [Durães and Madeira, 2006] proposed G-SWFIT and study the use of fault injection on Windows 2000. The same tool was later used to inject faults on Windows XP [Irrera and Vieira, 2014]. Due to its wide use on many applications, some studies have also focused on Linux. Kikuchi et al. [Kikuchi et al., 2014] studied the representativeness of failures caused through fault injection and Yoshimura et al. [Yoshimura et al., 2013] used fault injection to study how errors propagate on Linux. To overcome some of the challenges of using fault injection on large code-bases Van Der Kouwe and Tanenbaum [Van Der Kouwe and Tanenbaum, 2016] presented a hybrid framework and evaluate it on Linux and Minix. Some works have also focused on injecting faults on Android [Winter et al., 2015].

One of the challenges with fault injection is finding an adequate fault injector that can work on modern systems. In fact, many of the previous works rely on the same base fault injector but made ad-hoc changes to fit their purposes but typically do not make them available (e.g., [Swift et al., 2006; Depoutovitch and Stumm, 2010; Kwon et al., 2016; Cotroneo et al., 2018] are all based on the fault injector initially developed by Ng and Chen [Ng and Chen, 1999]). Other fault injectors, such as the one used by Durães and Madeira [Durães and Madeira, 2006] and Irrera and Vieira [Irrera and Vieira, 2014] have not been ported to recent versions and thus only work on old (and therefore no longer representative) systems. Given the very complex nature of fault injectors and modern systems, creating or updating fault injectors is not trivial and requires expert knowledge. As a result, works that intend to leverage fault injection to conduct further studies on complex systems are often limited to using deprecated systems (as is the case of OFP). Notwithstanding, Yoshimura et al. [Yoshimura et al., 2012] updated and refactored (and made available) the widely used fault injector from Ng and Chen [Ng and Chen, 1999], allowing its use on more recent versions of Linux. Nonetheless, even with a viable fault injector, considerable effort/knowledge is still required to set it up (e.g., many require recompiling the system/kernel) and implementing the whole experimental process (e.g., fault load, failure modes, monitors). These challenges and limitations are, in our opinion, the most prominent reasons why failure prediction on complex systems (such as OSs) has not been researched as much as it should.

Most existing work uses fault injection to directly assess the dependability of systems. However, for OFP fault injection is used to generate failure data to create failure predictors, similar to what Irrera and Vieira [Irrera and Vieira, 2014] did. This raises further challenges such as actually monitoring and timing fault activation, workloads, and failures. Additionally, as the goal is to monitor the target system, the behavior and performance of the experiments should be as consistent and repeatable as possible. This requires carefully planning and devising the testbed that will support the experiments.

A typical fault injection campaign requires a significant number of experiments to cover all relevant faults and achieve statistically representative results [Winter et al., 2015; Natella et al., 2012]. Moreover, fault injection can potentially corrupt the system, and thus the execution environment used for each experiment should be controlled. To address this, Banabic and Candea [Banabic and Candea, 2012] proposed a framework to expedite the use of fault injection techniques, taking advantage of parallelization approaches. Winter et al. [Winter et al., 2013] explore the use of simultaneous execution for multiple fault injections while Winter et al. [Winter et al., 2015] assess the impact/validity of running multiple fault injection experiments simultaneously. Schwahn et al. [Schwahn et al., 2019b] proposed a framework for fault injection that accelerates the process by executing experiments simultaneously and also avoiding redundant experiments. In fact, many of the techniques used to develop dependable software require executing large sets of experiments. *Software testing*, which consists of executing a program with the intent of finding faults [Myers et al., 2011], typically requires the execution of a considerable number of experiments [Hetzel and Hetzel, 1988; Kuhn et al., 2009]. Kapfhammer [Kapfhammer, 2001] parallelized his experiments for the problem of regression testing. The need and demand for parallelization are so relevant that there are studies on parallelizing test suites [Candido et al., 2017] and tools to ease the execution of unit tests in a parallel/distributed manner [Gambi et al., 2017]. A variation of testing, *robustness testing*, attempts to assess to which degree the program functions correctly in the presence of exceptional (e.g., range, invalid) inputs [Kropp et al., 1998]. Makai et al. [Makai et al., 2019] presented a case study on the methods used to achieve automatic regression and robustness testing on the CERN disk storage system, using GitLab-CI [GitLab, nd], which uses parallelization. Work in large-scale European projects also made use of parallelization to accelerate robustness testing [Fernández and Dullaert, 2018].

While some of the previous works actually parallelized experiments, others simply made use of multithreading. Such techniques, as well as virtualization, raise some concerns, such as if, and how much, interference exists between experiments [Schwahn et al., 2019a] and how it affects the results [Novaković et al., 2013]. They overlook the performance independence between experiments, as they were mainly focused on increasing throughput. Although some assess the profile of the experiments between sequential/simultaneous execution (e.g., are the observed failures similar [Winter et al., 2015]), they do not analyze the actual results/performance. While increasing throughput is indeed relevant, if simultaneous execution compromises the experiments then the results may be invalidated. For example, if the experiments are not isolated when assessing the performance impact in the presence of faults (i.e., the performance results vary between running 1 or N experiments simultaneously) then identifying variance due to the injected faults is not possible. While the need for isolation is easy to understand, achieving it is far more complex. As discussed in *Section 2.1*, so many factors in modern computers non-deterministically influence their performance that even identifying them all is not trivial. Besides differences between OSs and distributions, the architecture of the system also influences the load distribution (e.g., CPUs shared L# caches, multiple NUMA nodes). Moreover, to optimize resource usage and power consumption modern systems have schedulers that distribute tasks depending on the current demands and control the running frequency of the CPU depending on several factors (e.g., computational load, temperature).

2.3.2 Benchmarking

Benchmarking has been used in various research areas, with performance benchmarking likely being the most relevant domain (e.g., TPC and SPEC benchmarks), ranging from simple (e.g., HW/component) to very comprehensive benchmarks focusing on complex systems (e.g., databases [Vieira and Madeira, 2003]). They have contributed to the improvement of successive generations of systems. However, a common issue with existing benchmarks is that they often include/define a specific workload. How to assure that the results will hold in different contexts, including deviations inherent to real-world scenarios, remains an open issue.

Recently there have been several works on dependability benchmarking following different approaches (e.g., experimentation, modeling, fault injection) [Koopman et al., 1997; Vieira and Madeira, 2003]. Cotroneo et al. [Cotroneo et al., 2017] propose a dependability benchmark for Network Function Virtualization (NFV) to determine which virtualization, management, and application-level solutions provide the best dependability. Although not specifically a dependability benchmark, Nunes et al. [Nunes et al., 2018] propose a benchmark for Static Analysis Tools (SATs) that considers the use of scenarios to take into account the environment in which the SATs will be used. While relevant, the guidelines/process to create the predictive models is not very detailed and thus it becomes harder to actually compare alternative solutions (different approaches to training and assessing the models may lead to different and not comparable results). Other details are also overlooked, such as statistically comparing the solutions to assure the differences are significant.

Benchmarking ML algorithms is a known problem, typically addressed by using generic datasets that can be used independently of any system configuration. Notwithstanding, this is not a trivial task due to the wide variety of algorithms, techniques, and hyperparameters. Moreover, properly assessing and comparing the performance of the models in a consistent, fair, and repeatable way is not straightforward as various techniques can be used, which in turn depend on the characteristics of the problem/dataset. However, benchmarking OFP algorithms is different, as the failures observed likely depend on the particular system. This way, to understand the effective performance of an algorithm it has to be tested in the system where it will be used. In turn, this raises further challenges, such as assuring that the collected data can, in fact, be used to properly benchmark ML algorithms. Furthermore, the context in which the predictors will be used influences the choice of the best solution.

Salfner et al. [Salfner et al., 2010] were the first to define comparability of failure prediction approaches as a property that '*can only be achieved if two conditions are met: (i) a set of standard quality evaluation metrics is available, and (ii) publicly available reference data sets can be accessed*'. While the literature is rich in metrics, they are not independent from the data [Sokolova and Lapalme, 2009] and are often misused (e.g., using *accuracy* in imbalanced datasets). Over the years there have been some initiatives for building repositories for failure datasets (e.g., the Computer Failure Data Repository [Usenix and University, nd], Los Alamos National Laboratory [Laboratory, nd]) that publicly provide detailed failure data from several systems. Notwithstanding, they are still not enough to assess and compare failure prediction algorithms meant to be used in practice. In fact, to understand the effective performance of a failure prediction algorithm, it has to be tested on the system where it will be used.

Various works relying on ML for OFP have already been proposed. In recent years predicting disk failures has been an active research topic, even considering large-scale datacenters (e.g., [Zhu et al., 2013; Zhang et al., 2020]). Some work has also been done on predicting job-failures (e.g., [Jassas and Mahmoud, 2020, 2018]) where the authors also found a clear correlation between failing jobs and workload attributes. Irrera and Vieira [Irrera and Vieira, 2014] used fault injection to generate failure data and study the applicability and limitations of such a process in assessing and comparing failure prediction algorithms. However, from a benchmarking perspective, these works have several limitations, such as the fact that only a limited number of solutions were studied, and no guidance was given on how to select adequate metrics. Additionally, no specific benchmarking procedures are proposed nor validated to properly measure and compare the performance of the various solutions, and no objective comparisons are made taking into account the intended use of the models. This can be due to several reasons, such as the fact that properly assessing and comparing different models in a repeatable and consistent manner is not trivial.

2.3.3 Machine Learning (ML) and Dependability

Throughout the years, several techniques have been designed to increase dependability, which can be mainly divided into two large groups: *fault avoidance* (aims for fault-free systems) and *fault acceptance* (accepts and deals with the existence of faults). Due to the growing complexity and sheer dimension of software, traditional techniques such as code reviews and testing typically do not scale well nor in an affordable manner. Following the recent trend of using ML in complex problems, there has been some research on applying it to the dependability domain, as ML techniques can extract knowledge that would likely not be found otherwise. Alsina et al. [Alsina et al., 2018] used ML algorithms to predict the reliability of components and concluded that some achieve better results than the equivalent traditional technique. Nie et al. [Nie et al., 2018] successfully used ML to predict GPU errors in HPC systems. Alves et al. [Alves et al., 2016] use a large vulnerability dataset and concluded that some ML algorithms were able to predict all vulnerabilities using software metrics.

Although ML has been recurrently used in the dependability domain, most works are both limited to a small set of methods and are specifically implemented according to the scope of the experiment. This reduces the ability to compare results and even minor decisions or bugs can drastically influence the resulting models. As ML became widely used, various general-purpose tools have been developed in both open-source and enterprise contexts. For research purposes, enterprise solutions (e.g., *Feedzai* [Feedzai, nd], *DataRobot* [DataRobot, nd]) are not accessible, however, amongst the open-source tools, there is also no straightforward choice.

One of such tools, *H_2O.ai* [Cook, 2016], is a ML platform that has been gathering support (mostly at enterprise level). It includes several ML algorithms and techniques and a GUI, supported by several developers and a growing community. However, although it is currently open-source, it is owned by a small company, which can easily make it proprietary for profit. Additionally, its development practices are not very strict, resulting in hundreds of branches with failed tests in their Git repository, as well as some loose coding standards [H2O.ai, nd].

Weka [Eibe et al., 2016] is a well-known open-source ML tool and it contains a wide array of algorithms and techniques for data mining tasks. It is mostly known for its GUI (although it also provides a CLI and API) which allows the user to experiment and visualize different techniques. However, concerning research, it presents some disadvantages, which is why it is often used on an exploratory basis. Although its GUI mostly removes the need to code, it is not intuitive, and less common tasks are not easy to execute (e.g., combining multiple preprocessing techniques). Moreover, Weka is implemented in Java, which has a steep learning curve and often imposes a rigid and complex code structure. As a result, although it also provides an API, customizing and expanding it is not that trivial. Weka was initially developed in academia and is mostly maintained by a small team (although there are also third-party packages). Thus, its code is often not clean or efficient and the development and adoption of state-of-the-art techniques can take some time. Combined with the fact that Weka does not have a very active community, support is rather poor, which is only worsened by its documentation. In fact, throughout the years, Weka has been losing relevance (as highlighted by Saez et al. [Saez et al., 2017]) to more actively developed frameworks.

One of the most promising solutions currently available is *scikit-learn* [Pedregosa et al., 2011], which is a comprehensive library of ML algorithms and techniques. It has been widely adopted and thus has a large developer- and user-base. It is thoroughly documented which, combined with the active community, translates into good support for use and development. Its large number of developers allows it to steadily keep-up with advances in the state of the art, all the while ensuring peer-reviewing of its code towards standards and performance. It has an intuitive interface that is kept throughout the framework, allowing it to seamlessly integrate with other relevant ML packages (e.g., *Keras* [Chollet et al., 2015], *Tensorflow* [Abadi et al., 2015], *PyTorch* [Paszke et al., 2019]), and makes use of other scientific (e.g., *numpy*) and visualization packages (e.g., *matplotlib*). In fact, it has been gathering support even within the scientific community [Eshete and Venkatakrishnan, 2017; Saez et al., 2017]. However, it is made to be used only as an API and, although it provides several 'helper' methods, it still requires significant coding for a sound/comprehensive experimental process. Besides the programming effort, this requires that the user has considerable ML knowledge to properly conduct the experiments according to theory.

2.3.4 Online Failure Prediction (OFP)

OFP tries to predict incoming failures based on past failure data and the current state of the system. Transposing the concepts of OFP to ML terms, the monitored system variables are referred to as *features*. The set of values of those variables at a given time t is known as a *sample*, and whether a failure will or not occur for that sample is known as the *class, label,* or *target* of that sample. The work conducted in this thesis focuses mostly on *symptoms monitoring* methods, where the prediction is based on the continuous observation of the system state looking for symptoms (i.e., the values of the features) that may indicate the potential occurrence of failures. A key problem is that hundreds or thousands of features may be required to characterize the system state in a precise manner.

Despite the potential of OFP, it is still not widely implemented, partially because failures are rare events, and thus collecting data for training and testing new methods is a complex endeavor. To mitigate this, several studies have been conducted on the injection of software faults to generate representative failure data [Durães and Madeira, 2006; Irrera and Vieira, 2015; Irrera et al., 2015; Jordan et al., 2017; Pitakrat et al., 2018]. The use of virtualization and its impact on the generated data has also been studied [Irrera et al., 2013a]. Notwithstanding, conducting a proper fault injection campaign on a modern system is complex and time-consuming, and therefore it is still not a common practice. As a result, most recent work on OFP focuses on smaller components with a shorter lifespan (for which it is possible to find, or easier to generate, failure datasets, e.g., hard-drives [Zhang et al., 2020]). Various approaches relying on different sources of data (e.g., log files, system parameters) and ML have been proposed over the years [Salfner et al., 2010]. Among them, Salfner and Malek [Salfner and Malek, 2007] used a clustering approach based on the system state, alongside Hidden Semi-Markov Models to predict failure-prone states. Predicting disk failures has also been an active research topic (e.g., [Hughes et al., 2002; Zhu et al., 2013]), and, more recently, Zhang et al. [Zhang et al., 2020] addressed the problem of disk failure and replacement in large-scale data centers using a ML approach based on transfer learning, which outperformed traditional approaches. Jassas and Mahmoud focused on developing models to predict job-failures in large-scale cloud applications [Jassas and Mahmoud, 2020] and have also studied the workload features of similar job-failures and observed a clear correlation between failing jobs and workload attributes [Jassas and Mahmoud, 2018] (agreeing with our argument that failure data should be collected from the system where failure prediction is going to be implemented).

Meanwhile, OFP for complex systems (such as OSs) has become stale. Pitakrat et al. proposed a mechanism to classify and predict system events based on system logs [Pitakrat et al., 2014] and later they proposed an architecture-aware OFP approach that uses failure prediction at the component level and combines it with architectural knowledge [Pitakrat et al., 2018]. While relevant, these approaches are focused on well-defined hierarchical systems and accurate individual component failure predictors, which are assumptions that typically do not scale well to complex systems. Probably the most recent work that laid the groundwork for this thesis is the work of Irrera et al. [Irrera et al., 2013b]. The authors thoroughly study the use of fault injection as a valuable technique to generate failure data (using a G-SWFIT-based tool), which can be used for OFP. They use SVMs to predict two failure modes on specific workloads on Windows XP. Later they demonstrated the need to evolve predictive models [Irrera et al., 2014] and proposed a conceptual framework to achieve this [Irrera et al., 2015]. Additionally, they also analyze the influence of virtualized environments, as well as preliminary techniques to account for the dynamic nature of software. However, concerning the OFP problem, the authors did not conduct a thorough analysis of the different tasks, using a limited set of techniques and algorithms.

While OFP is in essence a classification problem (i.e., predict if a failure will occur based on the system variables at a given time) it can also be seen as a time series problem by considering the evolution of the system (e.g., values from previous instants) to predict an event in the future. Inherently, this means that there is a correlation between sequential instances (i.e., $t+1$ is related to t, *autocorrelation* [Bisgaard and Kulahci, 2011]). As OFP relies on monitoring multiple system variables, it is considered a multivariate time series. To use/extract this information several techniques have been developed (e.g., statistical/windowed feature extraction [Barandas et al., 2020]). However, OFP is not a typical time series problem, as it is not trying to forecast the values of the system variables, but rather an unwanted (discrete) event, whether or not the current state of the system is indicative of a failure in the near future. As its data are typically experiment-based, each experiment/run will have a different duration and there is also no sequence between them (i.e., no 'past' or 'future' of the system between experiments). Moreover, as failures will always be at the end of the experiment, approaches such as *out-of-sample* are not applicable. Ultimately, the current body of literature does not provide clear guidelines on how to combine these fields on a complex problem such as OS-level OFP, which in turn may lead to wrongful conclusions.

Overall, the main limitations of existing work on OFP are the following:

- due to lack of real failure data and the complexity of generating representative and realistic failure data most existing works focus on smaller components or on deprecated OSs. Moreover, among those that use fault injection the specific characteristics of using such techniques to generate failure data for OFP are ignored, which may ultimately compromise the results (e.g., false-predictors, such as *system uptime*, which do not have meaningful value in fault injection campaigns but may indirectly leak information of the process);

- most studies use the same prediction approaches, being limited to a few commonly used techniques and algorithms (mostly SVMs with some kernel variations) [Irrera et al., 2013b; Hoffmann et al., 2006; Jordan et al., 2017]. As there is no algorithm that is best for every problem [Wolpert and Macready, 1997], such a limited analysis does not take advantage of the plethora of algorithms and techniques, as well as the computational power that is available nowadays;
- the interpretation of the performance of the predictive models is limited, often focusing on individual metrics that may not even be adequate for the nature of the problem. Although there are many metrics and techniques to assess their performance, depending on the data and purpose of the predictors they should be conscientiously used, as they measure different things. By using inadequate metrics, the analysis and choice of the models may ultimately be wrong. Additionally, the ideal set of performance metrics also depends on the needs of the target system, which has also not been taken into consideration;
- studies are usually focused on a very specific experimental model, such as a limited faultload, a specific workload, or a very focused failure mode. Although when using a model the conclusions cannot be completely generalized (i.e., it is only possible to say that for the problem represented by that data, a model behaved as it did), by considering very specific contexts it is not possible to have an insight into how they generalize to other contexts (e.g., environments);
- finally, due to the ad-hoc nature of such studies and the nonexistence of standardized support for assessment, they cannot be compared or generalized. As such, it is impossible to compare approaches from different works, and any conclusions are constrained to the approach followed by the authors.

The lack of extensive use of ML techniques for OFP can partially be explained by the fact that some of the required techniques (such as the ability to quickly generate failure data and create sandbox environments) have only recently been used for OFP and still remain open issues. The fact that there are various complex interdisciplinary subjects involved (e.g., fault injection, OFP, ML) and no relevant document exists on how to combine them is also be a contributing factor, as such task requires extensive expertise and considerable effort.

2.4 Summary

Over the years considerable work has been done towards advancing the state of the art on both *dependability*, focusing on creating more dependable systems, and *artificial intelligence*, developing techniques that further allow extracting knowledge on evermore complicated problems. Notwithstanding, as shown in this chapter, both these fields are complex in nature and present several challenges when tackling new problems, such as OFP. Because OFP depends on complex techniques from both domains, and despite its potential, there is little research or documentation on how this can be achieved.

This chapter provided the necessary background knowledge and introduced the basic concepts required to understand the scope of this thesis, as well as the challenges and open issues it addresses. This chapter also presented and discussed existing relevant related work on the dependability and ML domains. It highlighted the various contributions that can be used to support the development of OFP solutions for complex systems and the major issues that still have not been properly solved.

Chapter 3

Techniques and Artifacts to Support OFP

Developing accurate failure predictors for modern complex systems is not a trivial task. It requires mastering both theory and practice from multiple research fields. As a result, tools and procedures to support research on OFP are not readily available, which is in fact one of the main constraints to the use of OFP.

Regardless of how many works demonstrate that OFP can be used to create failure predictors, developing predictive models requires training them on failure data from the target system. As it is not possible nor feasible to assume that failure data is available, it becomes necessary to somehow generate realistic failure data to overcome this issue. Fault injection has been accepted as the best alternative to generate failure data, but conducting a fault injection campaign on a complex system requires executing thousands of experiments to achieve statistical relevance. Moreover, a sound fault injection campaign requires several considerations (e.g., injection level, fault model) and fault injectors are typically very complex and system-specific (e.g., architecture-dependent). Fault injection also has limitations (e.g., representativeness) that should be taken into account and mitigated.

Although recent technological developments have led to an increase in computational power, implementing a testbed that can take advantage of the resources available without influencing the results is not trivial. To expedite the experimental process and reduce costs researchers often use modern techniques (e.g., hyperthreading) to run multiple experiments simultaneously, which relies on a premise of non-interference (i.e., executing multiple experiments simultaneously should not alter the behavior of the individual experiments). While some types of isolation are easier to attain (e.g., *software isolation*, the corruption or misbehavior of one experiment should not influence other experiments) others are quite difficult (e.g., *performance isolation*, where executing one or multiple experiments simultaneously should lead to similar results). This poses a challenge to the repeatability and validity of the experiments.

Using ML techniques to create predictive models on a new dataset is also an intricate endeavor. A thorough ML approach is a complex and problem-specific process comprised of several steps. Each of these steps includes multiple tasks, from exploratory and statistical analysis, to feature selection and extraction, model construction and selection, and performance assessment. The whole process must be correctly implemented, otherwise, the performance estimates obtained will likely not be representative. This requires mastering both theory and practice, as even a small mistake may inadvertently compromise the results. Various platforms have been proposed over the years to facilitate this process, but most cannot be easily customized, or have other limitations, such as restricted support or libraries. In fact, although comprehensive and well-established tools are available (e.g., Weka [Eibe et al., 2016]) there are still many researchers that rely on lower-level libraries (e.g., scikit-learn [Pedregosa et al., 2011]).

This chapter presents techniques and artifacts to support the development of predictive models for OFP. It provides guidelines on how to use fault injection to generate realistic failure data and how to configure a testbed that can leverage modern computational power to expedite the process without compromising the results. A comprehensive ML framework developed specifically for OFP and dependability research is also presented. It encompasses all the necessary steps for a thorough ML approach to create accurate and representative predictive models while ensuring the correct implementation of the different techniques and methods, with specific considerations for dependability and OFP.

3.1 Configuring and Deploying a Testbed

One of the most critical challenges for OFP is having enough failure data to create predictive models. Failures are rare events and thus failure data are often not available. Even if it were possible to gather such data from real systems, that would take years (due to the reliability of modern systems), and by then they would likely be outdated. While there are initiatives to build failure data repositories (e.g., Computer Failure Data Repository [Usenix and University, nd]) using such datasets is not enough as it does not take into account the system where the predictor will run. Hence, over the years fault injection has been accepted as a viable alternative to generate realistic failure data (e.g., [Irrera and Vieira, 2015; Cotroneo et al., 2019]). However, conducting a proper fault injection campaign on large code-bases is a complex and time-consuming task that requires executing thousands of experiments to be representative and achieve statistical relevance. In fact, this is a common issue across different subjects within dependability research, which aims at assisting in the development of dependable systems (e.g., robustness testing [Kropp et al., 1998]). To mitigate this, researchers often use heuristics to minimize the test set, possibly compromising the results obtained. While some works make use of distributed approaches to speed up the experimental process (e.g., [Parveen et al., 2009; Oriol and Ullah, 2010]), such setups are often not available for researchers.

Due to technological developments, there has been a considerable increase in computational power. Moreover, various techniques have been developed at the hardware-level, such as running multiple threads on a single core [Intel, nda], accelerating the processor for peak loads [Intel, ndb], and hardware virtualization [Intel, ndc]. To take advantage of the current computational power and accelerate the experimental process, as well as reduce hardware costs, several works have used multithreading to execute experiments simultaneously on a single machine (e.g., [Winter et al., 2015; Schwahn et al., 2019b]). However, this requires a completely automated testbed, which is not always easy to set up, especially when real-time monitoring functionalities and complex application interactions are required. Additionally, it also relies on the premise of non-interference, such as software containment and performance isolation. In short, executing experiments simultaneously should not alter the observed behavior and results. In this direction, virtualization techniques have been recurrently used for conducting experiments in the dependability domain (e.g., [Irrera et al., 2013a; Gambi et al., 2017]), as they facilitate the experimental process and its automation, as well as provide software containment (i.e., the misbehavior or failure of a VM does not influence other running VMs). However, performance isolation is not so trivial, as, by default, running experiments simultaneously will lead to lower individual performance. While this may not be a problem in some scenarios, it is often necessary to guarantee that the experiments are as consistent and repeatable as possible.

As a consequence of all the inherent experimental complexity and current technological solutions, devising and deploying an experimental testbed to assess the dependability of software systems is not straightforward. Furthermore, documentation, guidelines, and examples are not usually available, and thus properly implementing a testbed requires significant effort and expertise to identify all the relevant attributes, requirements, and implementation solutions. This frequently leads researchers to develop simplified testbeds focusing on their specific concerns, often not taking advantage of the computational resources available or neglecting aspects that may negatively influence the results of the experiments.

The work presented in this section attempts to overcome the aforementioned limitations by overviewing the concerns and requirements of a testbed for experiment-based dependability research. Additionally, we provide guidelines on how to create, configure, and attain the various testbed attributes. Although some of the commands presented in this section pertain to the Linux OS (as it is often the chosen platform for research) the concepts and guidelines apply to most modern OSs. These guidelines focus on achieving experiment isolation and complete automation so that multiple experiments can be executed simultaneously.

3.1.1 Drivers

OFP requires conducting large sets (e.g., thousands) of fault injection experiments to generate failure data. Such experiments are time-consuming, and given the computational power of modern systems, executing experiments simultaneously allows reducing both execution time and hardware experimentation costs (a prominent concern in research, as highlighted by Kanoun and Spainhower [Kanoun and Spainhower, 2008]). Notwithstanding, examples or guidelines on how to properly set up an adequate experimental testbed are not typically available. The main drivers that we establish that should be taken into consideration when devising a testbed for dependability experiments are the following:

- **Containment and Monitoring** – the main purpose of fault injection is to study the behavior of the system in the presence of faults, which may easily lead to corruption. Hence, each experiment should start with a clean version of the system. Additionally, it is also necessary to monitor the system under test to assess the impact of the injected faults. Due to the risk of an abrupt termination, it must be measured/stored every second.

- **Automation** – due to the potentially large number of experiments, the process needs to be as automated as possible to avoid the need for user interaction. It should be possible to automate the entire experimental setup: prepare/configure the host machine, execute the experiments (e.g., launch target machine, run fault injection/workloads, monitor/collect system metrics, detect failures), and terminate/clean the process (e.g., store/process/validate the collected data, restore host configurations).
- **Simultaneous Execution and Isolation** – the goal is to leverage the computational power available to expedite the experimental process by running multiple experiments simultaneously. However, identical experiments should produce identical results (i.e., the focus is on consistency and not peak performance). This premise should hold even if multiple experiments are running simultaneously, within fair use and share of the resources.

While the previous drivers are easy to understand, developing a testbed to achieve them is not straightforward. The following sections overview the most relevant concepts and provide guidelines and reflections on how to develop a fully automated testbed. They detail how to: *i)* configure the system to *attain performance isolation*; *ii)* leverage *virtualization for software containment*; *iii) monitor the experiments in real-time*; and *iv) automate the process*. A high-level illustration of the approach can be seen in *Figure 3.1*. Briefly, the host contains a controller that is in charge of handling all the workflow (e.g., isolating and controlling the experiments/VMs). Then, each VM sets up the required experiment configurations (e.g., shared folders), injects the faults instructed by the host, and executes the workload while monitoring the system.

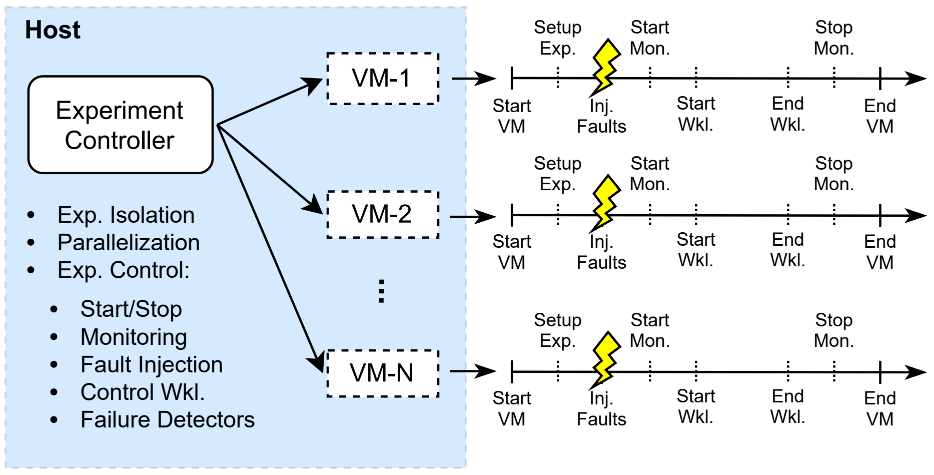


Figure 3.1: Experimental Process

3.1.2 Simultaneous Execution and Isolation

Running multiple experiments simultaneously allows taking advantage of modern computational power. However, for consistency and repeatability, it should not influence the results of the experiments. Performance isolation is not easy to attain and ultimately it will never be perfect without having dedicated hardware, running the experiments sequentially, or running them on separate machines. As an example, even if one 'guarantees' that a physical core from a multi-core CPU is exclusively dedicated to an experiment, some of the CPU components may still be shared (e.g., some architectures have shared L2 caches and most have shared L3 caches). Still, some solutions allow achieving decent performance isolation.

By default, when executing experiments simultaneously (e.g., through threads or processes), the OS scheduler will distribute tasks across the cores as it sees fit. Hence, to minimize interference and variations each experiment should be allocated to a fixed set of cores. However, one thing is to limit a process to a set of cores and another entirely different is to avoid any other processes from being scheduled to those cores. Two other concepts should also be taken into account. First, NUMA, a computer memory design where the memory access time depends on its location relative to the processor (i.e., a processor can access its own local memory faster than non-local memory) [Lameter, nd]. The cores dedicated to an experiment should belong to the same NUMA node (although in most architectures each CPU is a NUMA node, it is not always so, e.g., AMD Threadripper 1950X has 2 NUMA nodes). Second, logical cores of a given physical core share resources (e.g., L# caches) and should be used for the same experiment to avoid latency spikes. Hence, each experiment should run in a separate physical core.

A brief overview of the most relevant steps to achieve performance isolation can be seen in *Figure 3.2*.

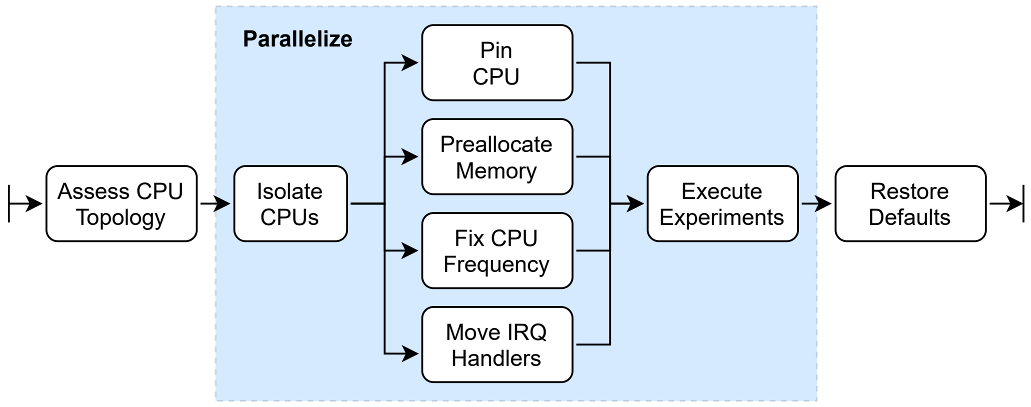


Figure 3.2: Performance Isolation Process

Isolating CPU Cores
To minimize interference, the logical cores required for the experiments should be isolated from the process scheduler. This can either be done using the *isolcpus* kernel parameter (a static approach, which will take effect at boot time)

or using *cset-shield/cset-(set)(proc)* functionalities (a runtime approach that creates/defines/isolates sets of cores, known as *cpusets*). While *cset-shield/cset-(set)(proc)* is often the recommended approach, as it provides more control over the isolation, it may not be able to move some processes already running on the intended cores. Nonetheless, both approaches may not be able to completely prevent kernel threads from being scheduled to the isolated cores.

The *cset-shield* program uses a concept of 3 cpusets [Ubuntu, ndd]: *i) root*: contains all cores (unshielded); *ii) system*: contains cores used for system tasks (unshielded); and *iii) user*: contains cores used for dedicated tasks (shielded). All userspace tasks will run in *system*, while *user* has nothing unless specifically set. However, *cset-shield* is only useful if one wants to isolate a single experiment (only one *user* cpuset). This way, it is necessary to use *cset-set* [Ubuntu, ndc] (create, adjust, rename, move and destroy cpusets) and *cset-proc* [Ubuntu, ndb] (manage threads and processes) to have finer control of cpusets. The concept is similar to *cset-shield*, but various *user* cpusets should be created (one for each experiment). Keep in mind that the logical cores in each cpuset must consider physical core affinities and NUMA, otherwise, errors or unwanted fluctuations may occur.

CPU Pinning
After isolating the intended cores, it is necessary to pin specific tasks/processes to them. Although there are various approaches (e.g., *taskset*), when using *cset-set* it is better to use its counterpart, *cset-proc*, which allows running a program on a given cpuset. Naturally, a cpuset should only be used for a single experiment at each time.

Preallocating Memory
Another technique to minimize interference between experiments is related to memory, which can have a noticeable impact on performance, especially in the case of latency-sensitive applications. Thus, preallocating memory for the experiments and increasing memory page size will help reducing memory access latencies and increase overall performance. Briefly, preallocation dedicates a contiguous area of memory so that it does not require to be dynamically allocated when needed. This provides several advantages, such as guaranteeing that no other process will use that memory, no allocation overhead exists, and no memory fragmentation occurs [Grimm, 2017]. When using VMs (which will later be discussed as a way to achieve software containment), a single configuration is often required to preallocate the memory for the VM.

Another improvement concerns the size of system pages. In short, the kernel needs to keep a table containing virtual-to-physical address mapping. Using small pages (e.g., 2KB) increases the total amount of entries in the index and therefore increases the time to look up/manage pages. *Hugepages* (e.g., 1GB) means that fewer pages will be required, reducing mapping, look-up, and maintenance overhead. Hugepages can be dynamically allocated during runtime. While most modern distributions have some high-level functionality that facilitates this, if the system has multiple NUMA nodes some precautions should be taken. First, it is necessary to define how many, and how, the experiments will be distributed across the NUMA nodes, and to calculate how many hugepages are required per node.

Then, one needs to check whether the pages are allocated as intended (not all distributions allocate them evenly across the nodes), otherwise, issues may later arise (e.g., starting a VM on a node that does not have enough hugepages left).

Limiting CPU Frequencies
Isolating cores, exclusively assigning them to experiments, and isolating memory may give the (wrong) idea that it is enough. Because energy efficiency is nowadays an important factor in the design of CPUs, cores can be completely turned off temporarily and their running frequency constantly changes depending on several factors (e.g., computational load, temperature). Additionally, specific techniques can also increase the frequency for peak load performance (e.g., Intel®Turbo Boost). This represents a challenge to systematic and repeatable experiments.

The goal is to disable/minimize variations of CPU frequency. While it could be tempting to set it for maximum speed, this may still be overridden (e.g., when reaching threshold temperatures). The more reliable way is to set the CPU frequency to its minimum, as it will not go below it. While this will force the CPU to work slower, often the focus is on comparison and repeatability and not on achieving the best performance possible. Thus, a performance penalty is usually acceptable, as far as it is the same for all experiments.

Minor Optimizations
Although often not as significant, other optimizations can also be carried:

- **Real-time Kernel** – using a real-time kernel may further reduce/eliminate latency and improve the predictability of thread scheduling. However, this typically requires manually recompiling the kernel and expert knowledge is advised. An alternative are low-latency kernels which are often available in the repositories and provide good real-time characteristics while keeping reliability [Ubuntu, nde].

- **Process Scheduler Tuning** – another approach to reduce latency is to set a process (the experiment) to use a real-time scheduling policy (e.g., *SCHED_FIFO*[Ubuntu, nda]). This will assign a higher scheduling priority to the process and therefore improve the predictability of thread scheduling.

- **NOHZ Full** – the kernel of modern OSs typically uses a scheduling clock that interrupts running applications to run a scheduler. Concerning Linux, version 3.10 introduced a full tickless mode (NOHZ full) that disables the scheduling clock when only one application is running on a CPU [LWN.net, nd].

- **Interrupt Request (IRQ) Pinning** – IRQs (hardware signals that trigger kernel interrupts) have an affinity property that specifies which cores can process them [Hat, nda]. This may be altered to avoid using isolated cores (i.e., cores assigned to experiments).

- **Offloading Ready-Copy-Update (RCU) Callbacks** – RCU is a lockless mechanism for mutual exclusion [Hat, ndb]. As a consequence, callbacks are often queued to be performed afterward. Seemingly, the *rcu_nocbs* kernel parameter avoids these callbacks on isolated cores [Hat, ndb].

- **Storage Optimizations** – intensive use of data storage (even Solid-State Drives (SSDs)) can lead to considerable wear and reduced performance. To minimize this, dirty blocks on disk should be periodically cleaned.

3.1.3 Using Virtualization

The use of virtualization is becoming a common practice. Besides being able to simulate machines running on different hardware and OS, it also has become the *de facto* approach to contain experiments (i.e., in theory, corruption of a VM does neither affect other VMs nor the host). Hence, it is a good solution for an automated testbed for failure data generation.

The problem is that, although VMs allow specifying the guest resources, they do not guarantee performance isolation [Jing et al., 2014; Matthews et al., 2007]. Bare-metal (i.e., *Type-I*) hypervisors may offer better isolation [Matthews et al., 2007] but they are not ideal for research (e.g., are mostly focused on enterprise solutions, limit the machine to just using virtualization). Additionally, because research is constantly evolving and often requires executing many experiments in varying environments, a prerequisite is that the chosen hypervisor must be flexible and easily scripted. While once again there are various alternatives, QEMU[QEMU, nd] (using KVM[KVM, nd]) is likely the most adequate option. QEMU may not be the easiest to get started, but it is highly flexible and can be seamlessly scripted. Moreover, QEMU provides several useful options for research and automation such as redirecting the output from the guest directly to the console and providing a separate kernel image. Additionally, combined with KVM, it allows for near-native performance.

After creating the necessary cpusets, QEMU can be initialized using the *cset-proc* command, specifying which cpuset to use. When pinning VMs to a cpuset one needs to be sure to leave one logical core for QEMU/system processing (e.g., if the cpuset has 2 logical cores, allocate just 1 to the VM). Concerning the other optimizations previously described, QEMU can be instructed to use hugepages (after properly allocating them) and which NUMA node to use (when applicable). Additionally, to improve latency, the VMs can be run using a real-time process scheduler. Kernel Same-page Merging (KSM), a memory-saving de-duplication feature that merges/shares memory pages used by KVM, should also be disabled. While this allows fitting more virtual machines into physical memory it may also introduce unwanted and unpredictable latencies when assessing/modifying the pages [Kernel, nd].

3.1.4 Monitorization

A key aspect is the capability of monitoring the target system. Although virtualization eases some problems, it also creates new challenges, such as collecting metrics/logs from the VMs. While in some scenarios it may be collected during execution and made available at the end of the experiment, for others (e.g., fault injection, which may abruptly terminate the VM), it should be continuously provided.

One common approach to monitor the state of a system is through its metrics (e.g., CPU/memory). This requires that a set of representative metrics (i.e., that characterize the system) should be chosen and collected. While it is possible to *a priori* select the most relevant metrics, this may require re-running the experiments if they are not adequate. A more generic approach is to monitor as many relevant metrics as possible (without impairing the system) and analyze them post-hoc, enabling possible knowledge discovery (e.g., metrics or synergies that would not have been inferred otherwise).

Most OSs provide native tools to gather system metrics (e.g., sysstat [sysstat, nd] in Linux). Still, most report the data in an unstructured format, and various tools are necessary to monitor the most relevant resources. Concerning free centralized solutions, while there are some well-known options (e.g., Munin [Munin, nd], Nagios [Nagios, nd]), it is not straightforward to access 'real-time' metrics (most are guided towards larger execution times). An adequate option is Netdata [Netdata, nd], a comprehensive, lightweight, and highly optimized tool (with negligible overhead) that provides real-time monitoring by default.

Concerning storage, although the easiest way to share data between the guest and the host is using shared folders, it may generate too much I/O activity. Instead, using the REST API provided by Netdata allows keeping the data in memory and saving it in batch. Notwithstanding, it is possible that, with specific configurations, other solutions may provide similar functionalities.

3.1.5 Process Automation

A testbed intended to run large sets of (simultaneous) experiments requires complete automation. This is often achieved through simple scripting (e.g., Bash) or programming languages (e.g., Python). However, more complex applications (e.g., interacting with VMs) may require more specific solutions. For example, *Expect* [Libes, 1995] is a TCL program that 'talks' to other programs, characterizing an interaction between user/program. Over the years it has been ported to various programming languages (e.g., Java [Gavrilov, 2018]).

Due to its flexibility, ease of use, and scientific packages, Python is one of the best languages for research. *Pexpect* (Python's implementation of Expect [Spurrier, nd]) allows automatically spawning and controlling applications. Mainly, it is comprised of two methods: *expect* and *sendline*. Briefly, *sendline* can be used to send commands to the application and the *expect* method awaits a successful match with the output given by the command. *Pexpect* also allows the use of conditional expects to deal with multiple possible outputs (e.g., a *sudo* command may or may not require a password).

3.2 Fault Injection to Generate Failure Data

Fault injection has become the *de facto* approach to generating representative failure data. For OFP, this allows overcoming one of the most prevalent limitations, the scarcity of data. However, fault injectors are difficult to implement or develop (or even to use) and thus research on failure prediction for complex systems (such as OSs) has become stale or relies on outdated datasets (e.g., [Irrera and Vieira, 2015]). Additionally, a typical fault injection campaign targeting complex systems also requires executing a large number of time-consuming experiments. In the end, properly conducting a fault injection campaign requires a fully automated and isolated testbed, which can be implemented following the guidelines described in the previous section. This will allow executing and monitoring large fault injection campaigns in feasible time without sacrificing representativeness.

Briefly revisiting some concepts, following the taxonomy proposed by Avizienis et al. [Avizienis et al., 2004], when the delivered service deviates from what is expected, it is known as a *failure*. A failure is due to a deviation in its state, known as an *error*. Errors are caused by *faults*, which can be internal (i.e., originated within the system boundaries) or external (i.e., originated outside the system boundaries). Fault injection intentionally introduces faults in the system to observe and assess how it handles in their presence. The (types of) faults to be injected constitute the *fault model*. Each fault type can only be injected in specific locations in the code (e.g., *missing initialization* can only be injected in instructions where a variable is being initialized), which are known as *fault candidates*.

In general, a fault injection environment is comprised of various components [Hsueh et al., 1997]: *i)* a *controller* (which controls the experiment); *ii)* a *fault injector*; *iii)* a *fault library/model*; *iv)* a *monitoring system*; and *v)* a *workload* to exercise the system. Although the concept of fault injection is easy to understand, conducting a proper fault injection campaign is a complex task that encompasses several design choices that must take into account the characteristics of the problem. Furthermore, using fault injection to generate failure data to support the development of predictive models for OFP requires specific considerations to assure the representativeness of the results. A high-level overview of the process can be seen in *Figure 3.3*. It is comprised of three main steps: *1) Preparation:* define and select the relevant components; *2) Generate Data:* execute the fault injection experiments; and *3) Process Data:* validate the generated data.

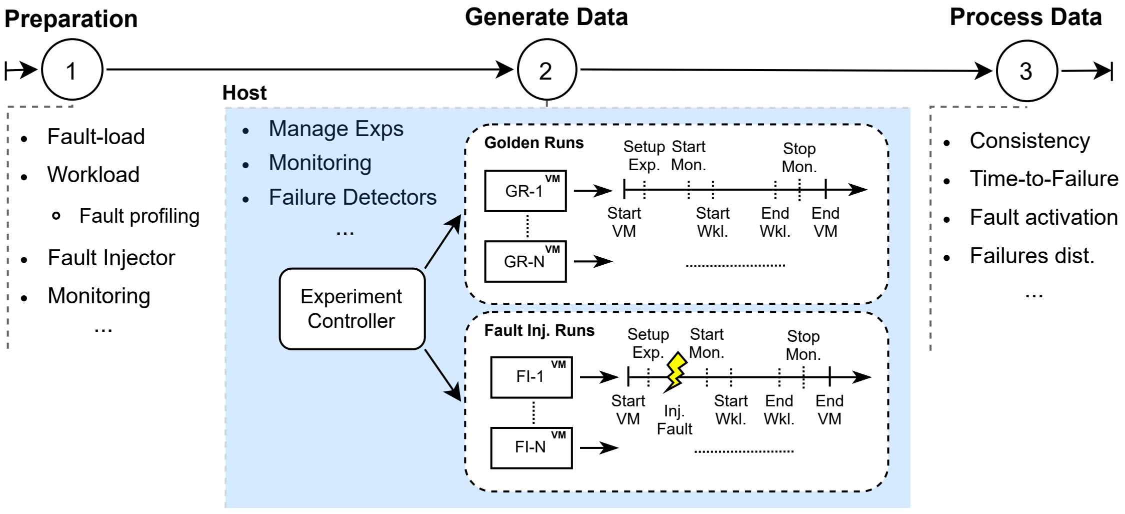


Figure 3.3: Data Generation Process

3.2.1 Preparation

The first step of a fault injection campaign is selecting and defining which techniques and configurations will be used. As can be seen in *Figure 3.3*, this includes the *workload* that will be executed, the *fault model* defining which types of faults will be injected, the *fault injector* to be used, the *failure modes* that will be monitored, and how the target system will be *monitored.*

Workload
A workload is needed to exercise the system and study the impact of the injected faults. As the workload influences the behavior of the system (with and without fault injection) it must be selected considering the technical needs of the system. The similarity of the workload to the operational scenario of the target system influences the representativeness and confidence one can put in the results. The duration of the workload execution should vary according to the needs of the system, but it must be at least long enough to allow the activation of faults and potential build-up to failure.

Fault Model
One of the premises of using fault injection to generate data is that they will be representative of real failure data. To achieve this, fault injection relies on injecting realistic faults (i.e., faults that have been proven to be recurrently made by programmers).

The definition of the fault model (i.e., the faults that will actually be injected) is one of the most relevant tasks of the first step. This model directly influences the validity of the results and therefore must be as realistic and representative as possible, otherwise, it can be argued that the failures (and subsequent predictors trained on such data) are not representative. Thus, the fault types ('what') and locations ('where') should be carefully defined. Moreover, depending on the requirements and analyses for which the data will be used, it may be necessary to consider additional factors (e.g., code coverage).

Injected faults range from low-level (e.g., bit-flips) to high-level (e.g., memory allocation) faults, which are nowadays commonly injected through software (also known as SWIFI) [Durães and Madeira, 2006]. The latter are the most relevant and intend to approximate real programming errors. They emulate various types of real faults, such as *assignment faults*, *control faults*, *parameter faults*, *omission faults*, and *pointer faults.* It should also be noted that not all injected faults lead to faulty behaviors (e.g., bugs inserted on a rarely/never executed path/condition will rarely/never produce an error). In fact, when targeting large code-bases the chance of injecting a fault in a path that will be executed is very low. Different approaches can be used to mitigate this, from injecting multiple faults in each experiment to profiling the execution of the workload to inject faults in places that are more likely to be executed.

Fault Injector
The next major task is selecting an adequate fault injector that supports the previously defined fault model. Faults can be injected at different levels, each with its own advantages and disadvantages. Typically, the injection can be made

at either the binary code (generic, less expensive, but loses context) or at the source-code (less generic and more expensive as it may require recompiling and source-code is often not available, but more accurate as it preserves context). To take advantage of both approaches, over the years there have also been some works on hybrid solutions (e.g., HSFI, which combines both source-code and binary-code fault injection [Kouwe and Tanenbaum, 2016]). Other works also explore injecting faults at other levels, such as intermediate-code [Lu et al., 2015].

Finding an adequate fault injector, that is both suitable and available, is not always easy. Especially when targeting complex systems (which typically have a large code-base), a trade-off must be considered among the viable alternatives, between ease of use, scalability, and representativeness.

Failure Modes
It is also necessary to define which types of failure are going to be considered. This is particularly relevant, as they often require developing and implementing specific failure detectors and will ultimately determine which types of failures can be predicted.

Failure modes can be based on historical or system-specific failures, or existing failure taxonomies (e.g., C.R.A.S.H [Koopman et al., 1997]). Some of the most common failure modes are *system crash* and *hang*, *performance deviation*, or diverse *system corruption* (e.g., filesystem). At a high-level, failures can also be divided into *fail-stop* (i.e., the system cannot continue afterward, e.g., system crash) and *non-fail-stop* (i.e., the system continues execution in a degraded mode, also referred to as *fail-soft* failures [Tipton and Krause, 2007], e.g., delayed cpu execution). While non-fail-stop failures may provide further granularity, it should be taken into consideration that multiple failures may occur within a given time-window.

Monitoring
The final major decision of this step is to determine how the state of the target system will be monitored. These data will later be used to create predictive models and thus should be thoroughly collected. Several sources of data can be considered, such as the system metrics or/and logs. Notwithstanding, different sources of data will influence the ML techniques that can/should be used to create predictive models. As an example, while numeric system metrics can be used with the most traditional ML algorithms (e.g., SVM), text-based data typically requires NLP pre-processing techniques and algorithms.

3.2.2 Generate Data

A fault injection campaign typically includes executing many (e.g., thousands) experiments with and without fault injection for a given workload. Given the number of potential experiments, the whole process should be automated.

For OFP, several types of runs can be considered, as depicted in *Figure 3.3*. Runs in which faults are injected are known as *fault injection runs*. They can be either *failing runs* if a failure occurs, or *non-failing runs* otherwise. To create predictive models that can distinguish between normal and failure-prone states it is also necessary to determine the baseline behavior of the system in the absence of faults (i.e., experiments where no faults are injected), which are known as *golden runs*.

Injecting faults may lead to the corruption of the target system. Thus, each experiment must start with a clean version (which is typically discarded at the end). Due to the stochastic nature of complex systems, and depending on the number of candidate fault locations, multiple experiments should be conducted for each fault. Moreover, it is not only relevant to assess whether a failure occurred or not, but also when the injected fault was activated. This allows measuring how long it takes between the fault activation and the failure occurrence and provides a hard timestamp from when the symptoms may start to manifest (i.e., symptoms cannot start prior to the fault activation). Injecting faults in complex systems is a difficult task, both due to their complexity and their size. It is often common that there are too many candidate locations and thus it is not possible to explore all of them. Different approaches can be considered to guide the process (e.g., profiling the workload and injecting only in code likely to be executed) to make it feasible without losing representativeness.

To detect failures, all the necessary failure detectors should be deployed and the target system continuously monitored. If some failure modes require a time-window to be considered (e.g., a hang failure is typically considered after the system becomes unresponsive for n seconds) the time of failure should be the first instant it occurred. Due to the complexity of modern systems, it is also possible that some failures may pass undetected by the failure detectors deployed. Depending on the purpose of the fault injection campaign, it may also be relevant to check for such failures (e.g., by monitoring the logs of the system). If non-fail-stop failures are considered, it is also necessary to define what will occur to the experiment when they are detected. Ideally, and for completeness, the experiment should execute until it ends (the workload execution duration is achieved) or a fail-stop failure occurs. This means that it is possible that some experiments may have multiple failures.

Ultimately, in the context of OFP, the main purpose of using fault injection is to generate failure data to create predictive models. This requires that for each experiment the system under test must be frequently monitored (e.g., every second) to collect the intended data. While extremely short intervals may provide a higher resolution (e.g., tenths-of-second) this adds a considerable overhead (i.e., each model would have to make predictions with the same interval) and it is likely that changes at the system level are not reflected at this granularity. On the other hand, larger sampling intervals may reduce the overhead, but may also result in losing relevant data. Thus, a trade-off between the goal of the system and its overhead/effectiveness must be considered. The most adequate tool to achieve this depends on the source of data that is used to construct the predictive models (e.g., system metrics can be collected through native applications, such as sysstat [sysstat, nd]), or third-party, such as Munin [Munin, nd])). To avoid delays and disruption, such requests should be made asynchronously and the time between requests should be monitored to ensure the correct resolution of the data. As fault injection may lead to an abrupt termination, these data should be safely stored (e.g., memory on the host, disk). Furthermore, because the amount of data may not be negligible, it is necessary to take some precautions to avoid overloading resources and inadvertently influencing the experiments. The logging of the timestamps should be as precise as possible to allow the data that will be used to train the models to be processed based on the timestamps of the different events in the experiments (e.g., fault injection, failures).

3.2.3 Process Data

As shown in *Figure 3.3*, the final step of a fault injection campaign comprehends validating the experiments. Typically, only a small fraction of the experiments will lead to failures. Still, they are not all suitable for OFP. Runs where the failure occurred immediately after injection (or activation) are considered *invalid* for OFP as they are not representative of residual faults (i.e., such faults would have likely been detected by traditional validation techniques), and would not provide enough data points to create predictive models. Furthermore, for some experiments it may not be possible to monitor the fault activation, and also, on rare occasions, the system monitor may be compromised, and therefore cannot be used for OFP. It is also possible that some failures are observed only after the workload ended. Although such experiments may be useful for several scenarios, if it is not possible to precisely identify when the failure occurred, it cannot be used for OFP.

The outcome of the experiments should also be validated, such as golden runs not having failures and whether their performance is still according to the established baseline (as no faults were injected). Experiments where no failure was detected but whose results or logs varied from the baseline should be analyzed to assess if a failure has passed undetected (eventually leading to implementing a new failure detector or updating an existing one). The integrity of the collected data should be analyzed (e.g., time gaps between requests) to validate if they can be used for OFP. Finally, the distribution of the faults and failures should be studied to assess their representativeness.

3.3 Propheticus: Machine Learning Toolbox

An adequate and thorough ML approach is complex and problem-dependent. It requires a deep understanding of the problem and its data to choose the more adequate techniques and algorithms. Moreover, it is very common that the available datasets present certain characteristics that make them difficult to process (e.g., imbalanced data, high/low dimensionality) and that must be adequately handled. Besides defining which techniques to use, there are many other concerns, such as how to estimate the performance of the models and how to properly compare different solutions. All these choices lead to the necessity of mastering both theory and practice, as implementing all the concepts and techniques is far from trivial. Furthermore, a single small mistake in the experiment can be enough to undermine the whole process.

As ML became widely adopted, various platforms have been developed (e.g., Weka

[Eibe et al., 2016], H_2O.ai [Cook, 2016]) that abstract its technical details. However, most cannot be easily customized or extended, contain small or limited libraries, have small communities and consequently evolve slowly, or abstract on a lower level and still require significant coding. Consequently, there is no commonly accepted tool within the dependability community, where each researcher uses ML differently, often limiting the extent of the experiment (e.g., both [Eshete and Venkatakrishnan, 2017] and [Sauvanaud et al., 2016] use only a single algorithm, based on similar work, without considering any other techniques). In fact, albeit comprehensive tools such as Weka are often used, there are still many researchers that resort to lower-level libraries, such as scikit-learn [Pedregosa et al., 2011] (e.g., [Eshete and Venkatakrishnan, 2017]). This suggests that, although such tools are adequate for certain purposes, they are not flexible or easily adaptable for many others.

We developed Propheticus[1], a framework that includes functionalities for all the steps in a ML work, from data analysis and preprocessing, to model assessment and comparison. It was created to overcome the limitations of existing tools towards research in the dependability area, which requires it to be flexible and adaptable to fit the needs of the users. Propheticus can be applied to a variety of problems (e.g., error detection, failure prediction, intrusion detection) to create models whose predictions can then be used to develop and deploy more dependable systems. To use it on a given problem, the user only needs to launch its Command Line Interface (CLI) as a normal Python script. Then, he can navigate through the menus to explore his data and execute the various tasks. Finally, the user can analyze and compare the results of different approaches to determine the best solution.

Propheticus emerged with the purpose of easing, automating, and assuring the workflow of a ML approach applied to the dependability domain, but can also be customized to fit unforeseen uses. It provides a data-centric approach so the user can focus on the problem instead of the implementation. Propheticus does not intend to replace (or be better than) other alternatives, but rather it was developed based on the unfulfilled necessity of a tool focused on research in the dependability community.

Although Propheticus does not attempt to remove the complexity of a ML work, some of its processes and rules are fairly standard. As a result, one of the main goals is that it should clearly define the workflow of the experiment, validating and providing useful feedback to the user along the process. Additionally, it also attempts to identify common issues that easily go undetected and may ultimately compromise the validity of the experiments (e.g., *data leakage*, a feature that is inadvertently directly related to the class/target but that has no meaning in the real-world).

[1]Code and demo available at `http://www.joaorcampos.com/propheticus`

3.3.1 Overall Architecture

A high-level overview of the architecture of Propheticus can be seen in *Figure 3.4*. The different processes are encapsulated in modules, which are briefly described in *Section 3.3.3*.

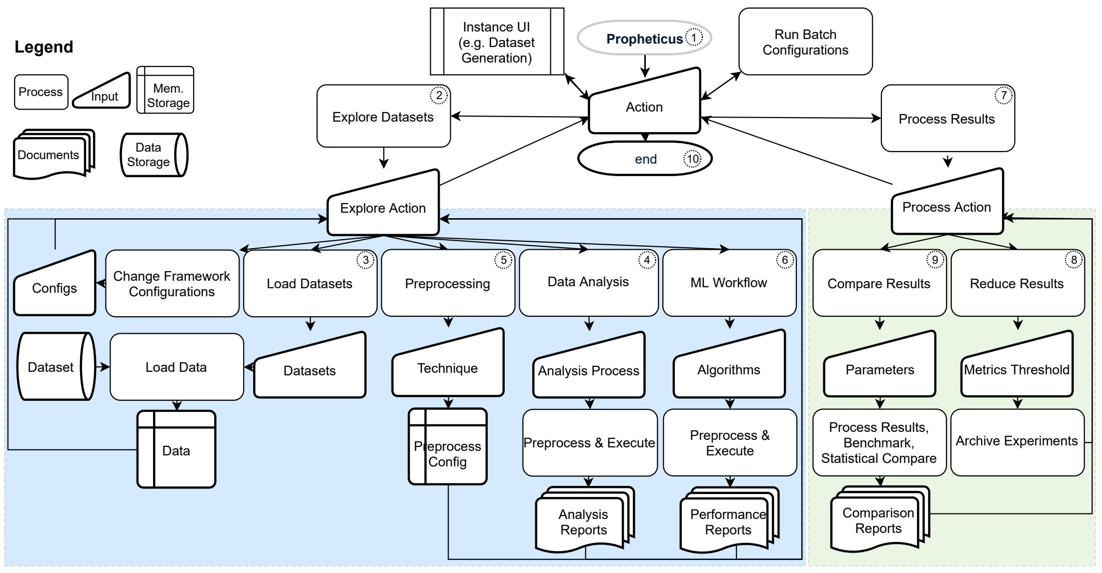


Figure 3.4: Propheticus High-Level Architecture/Workflow

At the moment, the use of Propheticus is based on a simple, yet comprehensive, CLI that allows the user to intuitively explore and flow through the different steps. To accommodate user-specific configurations and code, the framework must be instantiated (i.e., specific folders and files must be created to define the problem scope and configurations) for each problem. As one of the main requirements is for it to be flexible, Propheticus follows a hook-based design that allows the user to easily add functionalities (e.g., create datasets) to the CLI as well as binding to the core functions to add custom logic (e.g., running a specific task after each validation phase). Propheticus also implements a plugin paradigm for all the ML tasks so that new methods and techniques can be easily included in the framework (e.g., data splitting, performance metrics, algorithms).

The framework expects the input datasets in a specific structure, comprised of two files. The data file follows a straightforward structure: a simple table where the columns are divided into a set of features and targets, and where each row represents a sample that contains a value for each feature and its targets. The second file is the headers file, which must contain a JSON object identifying the details of the features.

For most functionalities, Propheticus creates reports that can later be used for analysis. These reports are stored on the hard drive and their filenames are hashed using all the configurations of the experiment. This allows generating the same report for different configurations without conflict.

3.3.2 Implementation

Propheticus is implemented in Python, a programming language that is flexible, easy to use, and prevalent in several research areas. Additionally, some of the most well-known ML libraries are implemented in Python (e.g., *scikit-learn* [Pedregosa et al., 2011], *Tensorflow* [Abadi et al., 2015], *PyTorch* [Paszke et al., 2019]). Propheticus leverages the research knowledge of open-source communities, and as such strongly relies on *scikit-learn* for most of its ML tasks (e.g., algorithms). scikit-learn includes several ML methods, is thoroughly documented, widely adopted and its community is quite large and active, resulting in regular updates that keep it up to date with stable state-of-the-art developments. Additionally, its structure is clear and simple, allowing the users to develop and seamlessly integrate with it. Other ML sources, such as *Imbalanced-learn* [Lemaître et al., 2017] (i.e., sampling techniques), *PyClustering* [Novikov, 2018] (i.e., clustering methods), and *SciPy* [Jones et al., 01] (i.e., statistics-related functionalities), are also used to complement the framework.

Propheticus runs on any system that supports Python. Currently, all processes are handled in memory, which requires the machine to have enough memory to hold the data. Other data management approaches will be considered in the future to improve scalability (e.g., online learning).

3.3.3 Functionalities/Modules

Propheticus acts as a standalone application and its workflow is centered on the interface module that controls all the process logic and calls the different modules. Although Propheticus is intended to be used iteratively, an ‘initial’ use would follow the numeric order identified in *Figure 3.4*.

The `Data Management` module handles the process of loading and preparing the datasets. It allows the combination of multiple datasets (e.g., different sources) as long as they share the same structure. Each dataset must have a headers file, containing the details (e.g., name, type) of each feature. Additionally, it is also possible to define other details (e.g., if a variable is categorical, which will be automatically encoded) or domain knowledge (not yet implemented) that may be used to improve the process (e.g., how to handle missing values).

The `Data Analysis` module encompasses the logic associated with exploring and analyzing the datasets. It contains functionalities for both descriptive and exploratory analysis. These generate reports that are stored on the hard drive for further analysis and comparison. Another module, `Data Preprocessing`, includes the logic required for preprocessing the data. It contains functionalities to select/exclude a subset of features, define which feature to use as *target*, or select only samples with certain values for given features. This module also includes the logic for dimensionality reduction and data sampling.

Probably the most important module, `Classification`, encompasses the execution of the whole experiment. The data is preprocessed according to the configurations and then passed to the selected ML algorithms. By default, it executes the same configuration 30 times under different random seeds (a number commonly accepted as adequate under the *Central Limit Theorem* [Hogg and Tanis, 2009]) and performs 10-fold [Borra and Di Ciaccio, 2010] stratified cross-validation (both configurations can be easily changed through the CLI). Several metrics are considered to assess the performance of the models (e.g., F_1-score, Informedness). These are computed fold-, run-, and experiment-wise. Upon finishing all the runs, the confusion-matrix, the ROC curve, and the Precision-Recall curve are generated, alongside a spreadsheet containing all the metrics and logs, which can then be used for further comparisons or analyses. As most algorithms contain various hyperparameters which can take a plethora of values, Propheticus allows fine-tuning them through grid-search. For this process, it uses a nested cross-validation approach (i.e., inner cross-validation is used to choose the parameters based on the training data) [Cawley and Talbot, 2010]. A similar, but smaller, module, `Clustering`, focuses on clustering algorithms. Seemingly, various clustering metrics (e.g., silhouette) and reports are stored on a spreadsheet.

After getting some insight into the problem, users frequently want to conduct several experiments in an exploratory manner (e.g., various algorithms, datasets). As executing all the combinations manually is not practical nor scalable, Propheticus allows defining a list of configurations to execute in batch, which is used by the `Batch Execution` module.

When all the experiments are finished it is necessary to compare the results and identify the solution that best fits the needs of the users, which is implemented by the `Experiments Comparison` module. One of its functionalities, *Reduce Results*, allows the user to (re)move (move to a separate folder) experiments that have performances for certain metrics under given thresholds (e.g., move all experiments with 0 recall). However, the more prominent functionality within this module, *Compare Results*, allows the user to choose the experiments that match given configuration parameters. This process generates a spreadsheet and various complementary graphs comparing the results (e.g., metrics, time complexity). Propheticus also explores the notion of application scenarios (a realistic situation of the problem at hand that depends on the criticality of the system) [Antunes and Vieira, 2015], which allows the user to compare and rank models based on a specific set of metrics. Finally, it also supports statistical comparisons between the experiments, informing the user if they are in fact different for a given significance level.

3.3.4 Instantiation and Configuration

Propheticus aims at being generalizable and configurable. This is achieved by having client-specific structures that allow the user to configure the framework to his needs while minimizing the need to alter the code of the framework. Moreover, it allows the user to have multiple projects under a single Propheticus installation. The customization of the framework is done mainly through three files: `InstanceConfig`, `InstanceGUI`, and `InstanceBatchConfiguration`.

For basic use (e.g., binary classification), it is not necessary to define any configurations. Still, framework- and problem-specific configurations (e.g., datasets location, binary/multi-class) can be specified in the `InstanceConfig` file.

The `InstanceGUI` file is not mandatory, but if it exists Propheticus will add a custom menu at the first level of the CLI to allow calls to user-specific menus (e.g., export datasets). Finally, the `InstanceBatchConfiguration` file can be used to generate/store a list of configurations of experiments that will be used by the `Batch Execution` module.

Propheticus also implements a plugin paradigm, which means that it is also possible to add new algorithms or techniques (e.g., dimensionality reduction, performance metrics) without modifying the code of the framework. This can be done by adding the new call details (e.g., package, method) to the existing list of the respective available methods. To integrate techniques that do not follow the scikit-learn structure, a wrapper extending the required corresponding interfaces can be created in a folder specifically for that purpose (e.g., `algorithms`).

Propheticus follows a hook-based design. This allows users to seamlessly have access or add functionalities to the internal methods (e.g., adding a custom functionality after each fold). This can be achieved by creating a new file named after the class it intends to access, using the prefix `Instance` (e.g., `InstanceClassification` to access the methods in `Classification`). Within that file, it is possible to hook into each method before (with the prefix `precall_`) or after (with the postfix `postcall_`) its execution (e.g., `postcall_runModel` would be executed after the method `runModel` in class `Classification`). Both `precall` and `postcall` methods have access and can alter the function parameters, and `postcall` also has access and can alter the returned values. If present, these hooks will be automatically detected and executed.

3.4 Summary

This chapter presented three contributions to support the development of predictive models for OFP. More precisely, it provided a well-defined process alongside instructions on how to use fault injection to generate realistic failure data. It also introduced detailed guidelines and reflections on how to configure and deploy an experimental testbed that can leverage modern computational power to expedite the experimental process without compromising the results. To assist in the process of exploring the generated data and assessing the performance and impact of different ML techniques, a comprehensive ML toolbox was also proposed. This toolbox was developed considering the specific characteristics of OFP and dependability research and includes all the necessary steps for a thorough ML approach while ensuring the correct implementation and combination of different techniques and methods.

The contributions presented in this chapter provide the necessary techniques and artifacts to enable the development of predictive models for OFP. Notwithstanding, using the data generated through this process is still not straightforward. For example, there is no thorough related work on how to properly use failure data generated through fault injection to develop accurate and representative failure predictors. These data, as well as the OFP problem itself, present certain characteristics that must be taken into account, otherwise, the experimental results may ultimately not be representative of the performance that the models will have in production. Moreover, due to the complexity of modern systems, developing accurate predictive models is not straightforward, and even tasks such as identifying and cataloging the failures observed may not be a trivial task. In the end, using data generated through fault injection on a new, unknown, problem is a complex task that often requires several iterations. Given the complexity and interdisciplinary knowledge required, detailed procedures or methodologies are needed.

The next chapter proposes a well-defined iterative methodology on how to use data generated through fault injection to develop accurate and representative predictive models for OFP. It takes into consideration the specific characteristics of fault injection and OFP, and each stage provides detailed guidelines, from generating and processing the data, to creating and deploying the predictors. Due to the exploratory nature of such a process, several feedback loops are also considered.

Chapter 4

Methodology for Developing Predictive Models

Developing accurate failure predictors for OFP is a complex task that includes several open issues, such as collecting or generating realistic failure data and developing accurate and representative failure predictors. In an attempt to overcome some of those limitations, recent works have combined fault injection and ML to create predictive models for OS-level OFP (e.g., [Irrera and Vieira, 2015]). However, such works focus mostly on fault injection and there is little research on how to properly use the generated data to create failure predictors. Furthermore, failure data obtained through fault injection, as well as the OFP problem itself, presents specific characteristics that must be taken into consideration when creating predictive models.

Due to the lack of guidelines or related work, most studies rely on standard techniques (e.g., *k-fold cross-validation*) which are often not adequate for the problem at hand. This may compromise the results, as they will not likely be representative of the performance that the predictors will have in production. A less direct consequence is that this can also affect the extent to which the problem is explored, which can result in having inadequate or unpredictable failure classes. Ultimately, this may lead to unrepresentative predictors that will not perform as expected when deployed in production.

This chapter proposes an iterative six-stage methodology that takes into consideration the particulars of using fault injection to generate failure data to develop predictive models for OFP. Each stage, which will be detailed in the following sections, defines a process and provides guidelines that should be considered, from generating and processing the data to ultimately creating and deploying the predictors. Given the complexity and exploratory nature of the problem, several feedback loops are included as it may often be necessary to return to a previous stage to expand the approach. A high-level illustration of the process can be seen in *Figure 4.1*. The proposed methodology is intricately connected to the contributions described in *Chapter 3*: it relies on the techniques used to generate failure data through fault injection, as well as the ML toolbox to explore the data and create accurate failure predictors using diverse ML methods.

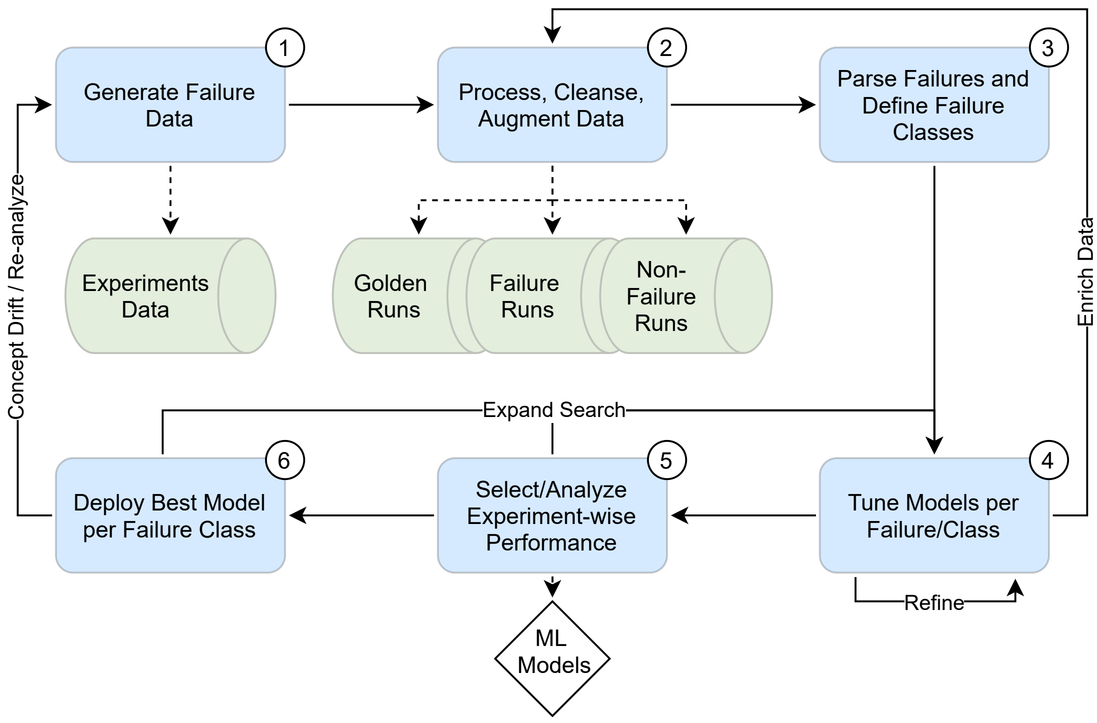

1
Generate Failure Data
2
Process, Cleanse, Augment Data
3
Parse Failures and Define Failure Classes
Experiments Data
Golden Runs
Failure Runs
Non-Failure Runs
Concept Drift / Re-analyze
Enrich Data
Expand Search
6
Deploy Best Model per Failure Class
5
Select/Analyze Experiment-wise Performance
4
Tune Models per Failure/Class
Refine
ML Models

Figure 4.1: Methodology to Create Predictors for OFP

4.1 Generate Failure Data

The first phase of the methodology, focused on generating failure data, comprises several decisions that directly influence the performance of the predictors. This section does not go in-depth on the fault injection process, as this has already been thoroughly documented in *Section 3.2*, focusing instead on the choices and considerations that will affect the ability to create predictive models.

Recapitulating, a fault injection campaign is usually comprised of five components: [Hsueh et al., 1997]: *i)* a *controller* (which controls the experiment); *ii)* a *fault injector*; *iii)* a *fault library/model*; *iv)* a *monitoring system*, and; *v)* a *workload* to exercise the system. It requires executing multiple experiments with and without injecting faults for a specific workload (experiments without faults injected are meant to establish a baseline behavior of the system). The workload influences the behavior of the system (and the potential failures) and therefore should be as similar as possible to the technical needs of the target system.

One of the most relevant definitions is which failure modes should be considered (that will later be mapped to different data classes). Several taxonomies have been proposed over the years (e.g., C.R.A.S.H. [Koopman et al., 1997]). While some failure modes are obvious (e.g., system hang and crash), nowadays most systems have advanced monitoring tools that also detect other less severe (e.g., non-fail-stop) failures, such as performance degradation. Thus, the failure detectors should be carefully developed, as it is possible to have multiple failures being observed in an experiment. To create accurate models, the logging of the timestamps (e.g., fault injection, failure detection) should be as precise as possible.

Another crucial decision is how the system should be monitored throughout the experiments. This has a significant impact on the performance of the models, as these are the data that will be used to create them. Although there are various alternative sources of data (e.g., logs [Salfner et al., 2010]) one of the most prominent approaches is through system metrics. This choice dictates which ML methods can be used to create the predictors (e.g., system logs require text-oriented techniques). Because the goal of OFP is to make a ‘continuous’ assessment and prediction, these data should be collected and stored in short intervals (e.g., every second). As discussed in *Section 3.2*, very short intervals provide more resolution (with a higher overhead) but it is likely that changes at the system-level are not reflected at this granularity. Larger sampling intervals may reduce the overhead, but may also result in losing relevant data. Thus, a trade-off between the goal of the system and its overhead/effectiveness must be defined.

4.2 Process, Cleanse, Augment Data

After conducting the fault injection campaign, it is necessary to process the generated data. Although a typical fault injection campaign comprises hundreds/thousands of experiments, most of them may not be useful for OFP. A representative activation/failure rate is accepted as being near 5% [Natella et al., 2010], which means that ~95% of the experiments do not lead to any failures. Even among the 5%, only a small percentage of experiments are suitable for OFP.

Experiments in which a failure occurred immediately after fault activation are not valid, as they are not representative of residual faults (i.e., they would have likely been detected through traditional validation techniques). Furthermore, as there is no time gap between the activation and the failure there are no symptoms to be learned. For a similar reason, experiments where it is not possible to register the fault activation should be excluded (as it is not possible to assure that there was an interval between the fault activation and the failure occurrence). Because OFP focuses on predicting failures during a typical system workload execution, one should only consider experiments where the failure occurs during the workload execution. In practice, only failures for which it is possible to precisely identify a timestamp during the execution of the target business process should be considered.

After selecting the experiments that can be considered, it is necessary to process the data before creating predictive models. Every experiment should have the same number of features. However, due to the stochastic nature of complex systems (e.g., OSs), some experiments may contain transient metrics, which should be discarded for sake of consistency (it is not practical to use datasets with a varying set of features). Metrics for which there is no variation in any of the experiments should also be removed to reduce complexity, as they add no value to the models. Another relevant concern due to the experiment-based nature of the generated data is to identify and remove all ‘false-predictor’ features, that is, features that may contain information related to what is meant to be predicted but have no meaning in real systems (e.g., given the repetitive nature of the experiments, metrics such as *system uptime* may leak information that does not have the same relevance on a real system). The data should be normalized (e.g., using Z-score) to accommodate metrics that have different scales.

Given the inherent complexity of modern systems and workloads and the fact that OFP is also a time series problem, it is likely that the absolute values of the system metrics are not enough to create accurate predictive models. Several techniques are available to augment and enrich the dataset considering the sequential nature of the problem (e.g., 5s-average) [Barandas et al., 2020]. Notwithstanding, this should be done with some caution as it will multiply the number of features (and thus increase the complexity) by the number of intended indicators.

Each sample should be labeled according to the definition of the OFP problem, as proposed by Salfner et al. [Salfner et al., 2010]. *Figure 4.2* (which shows the normalized values of all the collected system metrics of a failing experiment) illustrates how the label of the samples of a failure experiment evolves over time (for $\Delta t_l = 20$ and $\Delta t_p = 30$, $t_{failure}$ is the time of the first failure, ~110 seconds after the workload began). In the end, this leads to different datasets for each pair of $\Delta t_l, \Delta t_p$.

As shown in *Figure 4.1*, this stage divides the experiments into three categories: *failure* runs (experiments where a fault was injected and a failure was observed), *non-failure* runs (a fault was injected but no failure was detected), and *golden* runs (the baseline experiments, without fault injection).

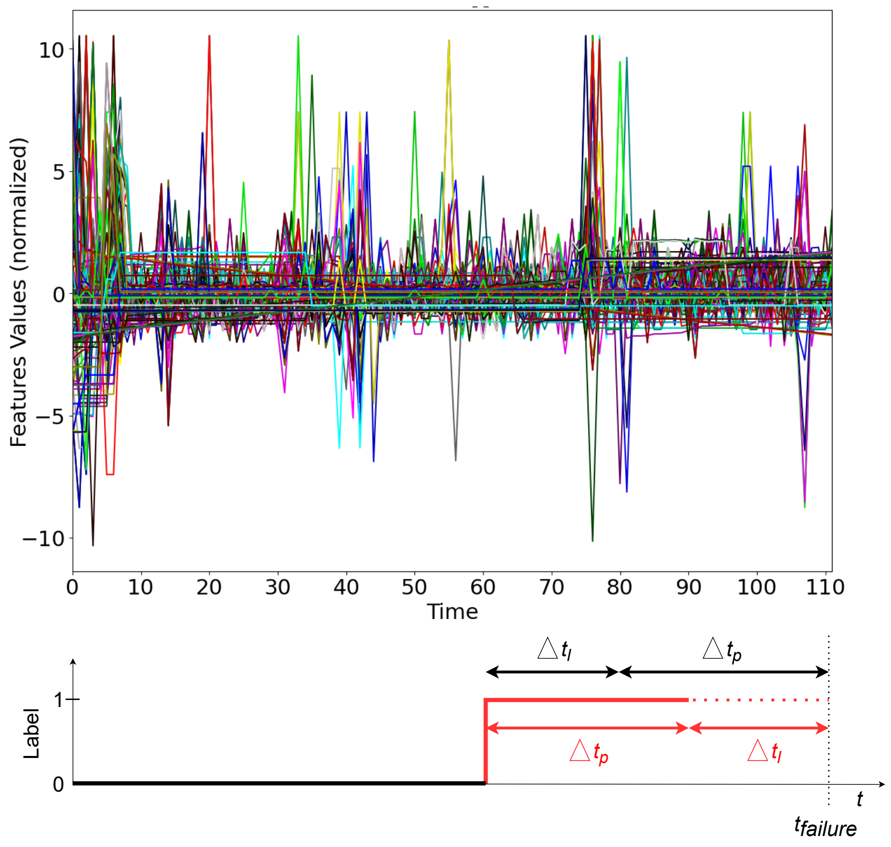


Figure 4.2: Experiment Labeling [Salfner et al., 2010] for $\Delta t_l = 20$ and $\Delta t_p = 30$

4.3 Parse Failures and Define Failure Classes

The severity of a (predicted) failure may determine which preemptive measures should be taken. Thus, it is relevant to distinguish between different 'classes' of failure (e.g., fail-stop and non-fail-stop failures). While this may be easy to determine for a simple system, for a complex one (with many different types of failures) it is not trivial. After selecting which failure experiments can be used for OFP, it is necessary to define the classes of the problem (i.e., which classes will the models try to predict).

Over the years, several failure taxonomies have been developed (e.g., C.R.A.S.H. [Koopman et al., 1997]), which can be used as a starting point. At a high-level, failures can be divided into *fail-stop* and *non-fail-stop* (similar to the distinction between *catastrophic* and *non-catastrophic* failures considered by Powell et al. [Powell et al., 2001] and Crouzet and Kanoun [Crouzet and Kanoun, 2012]). Fail-stop failures have severe implications, that abruptly halt the execution of the system (e.g., the OS becomes corrupt and the system crashes or reboots; the OS becomes unresponsive, i.e., hangs, and must be terminated by force), while non-fail-stop failures (also referred to as *fail-soft* failures [Tipton and Krause, 2007]) typically have less perceptible consequences that allow the system to continue execution, often in a degraded mode (e.g., memory failures such as *segmentation fault* or execution failures such as *invalid opcode* that the system can tolerate). As there are usually few types of fail-stop failures, it is acceptable to consider each as a class of the problem. On the other hand, in complex systems (which can be comprised of several components), multiple types of non-fail-stop failures can be easily identified, making it difficult to create models that can predict each one individually. There will typically be few examples/failures per failure type, and different failures with a similar root cause will likely exhibit similar symptoms, making the classification problem too complex/impossible. To make this process feasible, the most intuitive approach is by grouping different types of non-fail-stop failures into higher-level failure classes based on their (likely) root cause/similarity (e.g., *segmentation fault* and *Resident Set Size (RSS) counter error* failures are both memory-related). Another alternative used in ML for similar tasks is *clustering analysis*, which can be used to identify classes with similar characteristics through clustering algorithms [Gan et al., 2013].

Leveraging existing literature, the various failures must be assessed to determine why they occur. Although partially 'vague', this process is always required when assessing and exploring the different failures of a system. Furthermore, while it is possible to simplify the problem by creating large groups of failures (e.g., simply grouping fail-stop/non-fail-stop failures) this probably does not perform well due to the fact that different types of failures likely have distinct symptoms. When multiple failures occur simultaneously, their combination should be considered as a whole (e.g., if *kernel bug* and *invalid opcode* failures occur simultaneously they should be interpreted as a single failure *kernel bug + invalid opcode* and not as two independent failures). Moreover, a threshold should be defined during which sequential failures/alerts are considered as a single event (e.g., sequential failures logged 1 second apart are likely related and should be considered as a single failure).

Although these failure classes will be further validated throughout the process, they establish the baseline used to guide the development of the predictors in the following phases of the methodology.

4.4 Tune Models per Failure/Class

Creating accurate predictive models requires well-defined failure classes and an adequate method to estimate their performance. This stage focuses on properly estimating the performance of the predictors and identifying/organizing the various failures into classes that can be predicted (in the form of an iterative cycle). As often the raw data are not enough, this stage also considers a feedback loop to *Stage 2* to enrich/augment the dataset when necessary. Next, we discuss three key aspects: how to *identify and define* the failure classes, how to *train and test* predictive models, and how to leverage ML techniques to *develop accurate predictors.*

Identifying/Defining Failures Classes
Defining failure classes in complex systems is not trivial. Failures may be wrongly classified in an incorrect class and therefore affect the performance of the predictors. Additionally, given the complexity of modern systems, it is possible that some failures cannot be predicted (e.g., there are no relevant symptoms) and thus should not be considered. In practice, the failure taxonomy defined in the previous stage must be validated/refined.

Two approaches can be considered to achieve this: *i) top-down*, starting with the high-level taxonomy defined in the previous stage, try to identify which experiments are being mispredicted in each class (and exploring to which, if any, class they belong to); or *ii) bottom-up*, create predictors for each type of failure (e.g., *segmentation fault*, experiments with identical failures must be predicted by the using the same model) and iteratively combine with other types of failure with common characteristics (e.g., using the same $\Delta t_l, \Delta t_p$) guided by the high-level taxonomy previously defined. While the first approach may seem 'quicker' (as it already starts with an initial categorization), identifying which experiments in which classes are wrong is not straightforward. Combined with the possibility that some failures may not be modeled/predicted this may become an exhaustive, time-consuming, search. On the other hand, the second approach allows establishing a baseline performance for each type of failure and iteratively refining the classes of the problem according to the theoretical taxonomy. Moreover, it allows identifying failures that cannot be modeled and (more) easily determining misplacements from the previous stage. Notwithstanding, the classes identified through this process must be logical from the perspective of the system, and divergences from the previously defined taxonomy should be understood/validated. To illustrate the process, let's take a brief example from the experimental evaluation that will be presented in *Chapter 6*. Using the *bottom-up* approach, the first step was assessing/developing accurate failure predictors for each type of failure (e.g., *segmentation fault*). Afterward, failure types whose best models had similar characteristics (e.g., lead-time, prediction-window) were iteratively combined into higher-level classes (e.g., combining *segmentation fault* with *RSS counter error*, and then with *corrupt kernel paging* as a *memory*-related failure class). Each of these associations was validated against the initial taxonomy/classification to assess whether the selected failure types shared a common likely root cause. This process allowed identifying some issues with the initial taxonomy, when one of the failure types was being included in a different high-level class than what was initially defined. This was validated by inspecting the failure in more depth, and concluding that the initial analysis was incorrect (e.g., one of the failures at *ext4_evict_inode* was being considered as *kernel*-related but it was in fact related to memory management). Additionally, this also allowed identifying three failures (all due to the same fault) that could not be predicted. These failures occurred precisely when the workload was ending and it was likely due to some corruption that did not exhibit any symptoms.

Training/Testing Predictive Models
Given the specific characteristics of both OFP and failure datasets generated through fault injection, creating accurate predictors entails several considerations, in particular, regarding how to assess their performance (i.e., how should the data be split for training and testing). As discussed in *Section 2.2*, OFP is a classification (or regression) problem and, at the same time, a time series problem. However, it is not a typical time series, as instead of forecasting values of some variables, it tries to predict whether the current state of the system indicates that an event (failure) will occur in the near future. Additionally, failure data are typically experiment-based, where the failure occurs at the end of the experiment, and there is no sequence between experiments (i.e., no 'past' or 'future' of the system after each failure). As a result, although there is considerable literature on how to assess models for classification and typical time series problems, OFP falls in a gray area where there is no related work that supports its characteristics. Furthermore, existing techniques are not adequate and may even provide a wrongful performance estimate (e.g., as will be shown in *Section 6.3*, cross-validation provides an overoptimistic estimate).

To address this issue and properly assess ML models for OFP, we propose *Experiment-wise Leave-one-out Cross-Validation (ELOOCV)*, a novel process based on the leave-one-out cross-validation approach (where the test set is comprised of a single sample and the remainder data is considered for training, and iterates over the whole dataset [Alpaydin, 2014]). In short, ELOOCV considers every experiment except one for training and then tests on the held-out experiment, iterating over all the experiments. A simple illustration of the process can be seen in *Figure 4.3*. This process provides estimate assurances similar to that of cross-validation techniques with the addition that it takes into consideration the experiment-based nature of data generated through fault injection and the potential knowledge leakage of other approaches. In this way, predictions are only made on entire unseen experiments, which more realistically represents the environment where the predictor will operate. While this process can be more time-consuming than a traditional cross-validation approach, it is the most effective way to have an adequate estimate of how the model will perform against unseen experiments. In the unlikely event that one manages to generate a large volume of failure data for each failure class, it is possible to adapt the proposed approach to leave multiple experiments for test (if there are in fact enough and representative experiments for training). Notwithstanding, the premise that no sample (either failure or non-failure) from the experiments in the test set should be used for training must hold.

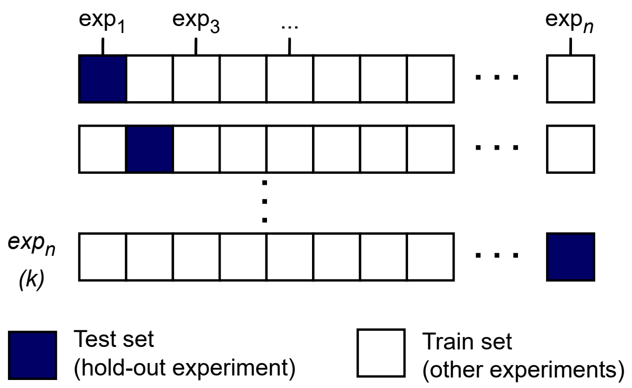


Figure 4.3: Experiment-wise Leave-one-out Cross-Validation (ELOOCV)

When trying to create models for a specific failure class (target failure class), what happens is that only the experiments where failures of the target failure/class were observed are considered as ‘positive/failure’ (labeled according to the approach proposed by Salfner et al. [Salfner et al., 2010] and illustrated in *Figure 4.2*). All other experiments, even other failure runs, are considered as ‘negative/non-failure’ because the model is meant to predict a specific class of failures. Thus, a relevant decision is to choose which experiments are considered for training. Logically, all failures from the target failure/class (except those in the test set) should be used for training. But concerning the remaining experiments (i.e., golden, non-failing, and other failing runs), there can be multiple approaches. The naive solution is to include all the experiments. However, this introduces several challenges as other failures may have similar symptoms (e.g., it is plausible that some non-fail-stop failures may have symptoms identical to fail-stop, as many non-fail-stop failures ultimately lead to fail-stop failures) that may lower the ability of the algorithms to model the target failure/class. Additionally, as the number of positive samples for each failure/class is typically low, sampling from such a diverse dataset to balance the classes distribution (which is often necessary for imbalanced datasets) may lead to high variance in the results (e.g., one iteration may select samples mostly from golden runs while a second may select only samples from other failing runs). To address this issue, heuristic/algorithm-based sampling techniques should be considered to reduce the variation between multiple executions. Another approach, which intends to simplify the task of modeling the target failure/class and improving the repeatability/stability of the results, is to consider only the golden runs for training (besides the experiments from the target failure/class obviously). It is worth clarifying that this process concerns only the data used for training, inside each loop of ELOOCV. ELOOCV should consider all the experiments (i.e., every experiment should be considered for test in some fold) to have a realistic estimate of the correct/incorrect predictions for each failure class.

Creating Accurate Failure Predictors

As vastly shown in the literature, different ML methods create the best models for different problems. Thus, to develop accurate failure predictors, the set of algorithms and techniques to be studied should be comprehensive and able to handle the characteristics of the dataset (e.g., data imbalance). First, an exploratory study should be conducted for the various methods to exclude algorithms or techniques that do not meet a minimum acceptable performance (while this may depend on the problem, a typical minimum could be predicting at least 50% of failures and no more than 25% of false-positives). Then, the parameters of the methods should be thoroughly explored. Although this may lead to a considerable number of combinations, the most likely scenario is that after the initial exploratory study it is possible to focus on a much smaller set of methods.

The parameters of OFP (i.e., $\Delta t_l, \Delta t_p$) should be studied carefully. It should be noted that the values of $\Delta t_l, \Delta t_p$ for each failure/class depend on the Time-to-Failure (TTF) of the experiments (i.e., the time between fault activation and failure), as it is not possible to use $\Delta t_l, \Delta t_p$ that are too close/larger than the TTF of the experiments (as this would leave no symptoms to model). Several metrics can be used to measure the performance of the models, depending on the intended use of the system. However, given the imbalance in the data, the use of metrics that consider both the positive and the negative predictions, such as Informedness (how consistently a predictor predicts the outcome of both the failures and non-failures) or F_2-score (which gives double the importance to recall compared to precision), is advisable.

When using the *top-down* approach to define/refine the failure classes, we need to explore the misclassifications (i.e., which failures/experiments were not predicted or predicted as a different type) for the various failure classes. It is necessary to assess if the misclassifications occurred because the experiments were included in the wrong high-level failure class (and to which class they belong instead), or if they refer to failures that cannot be predicted (e.g., no relevant symptoms prior to failure). In practice, if there are several failure classes and many misclassified experiments, this approach can easily become too cumbersome. Alternatively, for the *bottom-up* approach it is necessary to identify common denominators, by analyzing the parameters of the best models for the various failures/classes. These similarities should be studied and, if possible, the respective failures/classes should be combined. This is an iterative process: after identifying similarities between failures/classes, they are 'grouped' (i.e., considered as a class), and the whole process is repeated until it is no longer possible/beneficial to combine more failures/classes (as the example previously provided, the best models to predict *segmentation fault* failures presented similar characteristics with those of *RSS counter error* and thus the failure types were grouped; afterward they were also grouped with *corrupt kernel paging* into a high-level *memory*-related failure class). Each iteration of the process should take into account the initial high-level taxonomy and, when the resulting combinations differ from it, they should be explored to understand why. If, however, it is not possible to find acceptably accurate models for the failures in the dataset, it may be necessary to return to *Stage 2* to enrich it with more data.

4.5 Analyze Experiment-wise Performance

The previous stage focused on validating the failure classes of the problem and identifying the best techniques and parameters to create accurate predictors. However, the process focused on sample-wise prediction (i.e., predicting samples independently, regardless of their impact on detecting failures or raising false-alerts), which does not consider how many failures would have been predicted or how many false-alerts would have been raised. Real-world scenarios are concerned with more practical analyses, experiment-wise (i.e., how many failures are actually detected, and how many false-alerts are thrown).

In practice, it does not make sense to take actions based on a single alert, but rather on a sequence of closely time-related predictions. Thus, it is necessary to define how many alerts are required in a given time window (incidence) to consider it a 'failure prediction' (e.g., 4 alerts in the previous 5 seconds, as illustrated in *Figure 4.4*). Besides providing more consistent behaviors (against a single alert), it may also provide some tolerance to variations (e.g., a 'non-failure' prediction in the middle of alerts). This concept also allows tuning the sensitivity of the model per failure mode and according to the needs of the system. Ultimately, the goal is to assess how effective the models would be in predicting the various failures, from the perspective of the system or system administrator. Besides the obvious requirement of predicting incoming failures, it is also necessary to interpret the impact of the various incorrect predictions (e.g., a 'healthy' system state that is predicted as failure prone, or a fail-stop being predicted as non-fail-stop).

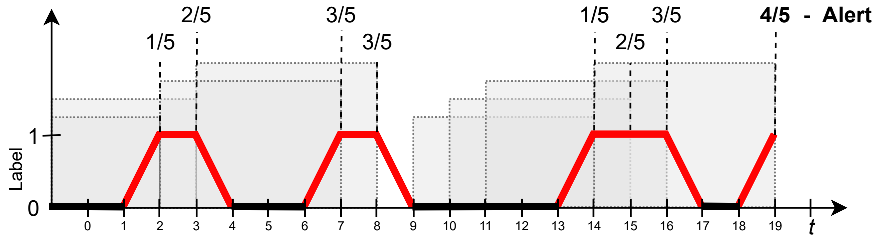


Figure 4.4: Experiment-wise Failure Prediction (4 alerts in the last 5 seconds)

While the ideal would be to perfectly differentiate between non-failure and every failure class, due to the potential relationship between some failures this is unlikely to occur (e.g., the symptoms of some non-fail-stop failures may be similar to some fail-stop failures, representing a sort of 'preceding' symptoms). Thus, the goal becomes to create models that can accurately predict failures with few false-alerts (as it is not acceptable to regularly interrupt execution unnecessarily) and that minimize the chance of underestimating the severity of the failure (and subsequent preemptive measures). As an example, a fail-stop failure that is incorrectly predicted as non-fail-stop may put the system at risk because it may not take the adequate preemptive measures, while a non-fail-stop being predicted as fail-stop is not as problematic, as the 'only' downside is that the system will likely take more conservative measures. Furthermore, if the predictors do not have a high count of false positives it is also possible to establish a second prediction incidence to start preparing repair mechanisms (and thus reducing the time to trigger mitigation techniques if it in fact predicts a failure).

Although sample-wise performance (from the previous stage) allows ranking the various models, it is not guaranteed that the absolute best will also be the best for experiment-wise prediction (e.g., they may have few false positives but scattered over multiple experiments such as golden runs). Thus, the top N models for each failure/class, ideally based on different techniques, should be analyzed. Still, even then the set of chosen ML techniques may not be enough to create adequate experiment-wise predictors. If this is the case, one should return to the previous stage and expand the set of ML techniques and/or parameters. If single algorithms fail to create accurate predictors, exploring heterogeneous ensembles (i.e., ensembles composed of different learners) may lead to better results by combining distinct learners with different biases. It is worth noting that when returning to the previous step it is no longer required to re-analyze the taxonomy of failures: the results previously obtained, whilst not good enough, already allowed identifying the failure classes.

4.6 Deploy the Best Model per Failure Class

The last stage of the methodology comprises deploying the predictors developed in the previous stages and monitoring their performance throughout time. In short, if at runtime any of the models matches the intended incidence (i.e., n alerts of incoming failures in a given time window), the system should trigger preemptive measures to avoid or mitigate the consequences of the incoming failure. Still, it may encounter some failures that cannot be predicted.

The models created in the previous stages allow predicting (known) incoming failures. However, making runtime predictions requires additional considerations. For example, at runtime, the set of collected metrics may sometimes not directly match the features used by the model. In this case, it is necessary to select only those used by the model, and when they are not available, register the event and skip the prediction (as it is not possible to make a prediction without the necessary data). If this occurs for several (sequential) seconds, then likely the system is also failing and preemptive measures should be taken. Given the complexity of the systems (and inherent limitations/coverage of the training data), it is also possible that some other failures may be missed. The logs and metrics from those failures should be saved, analyzed, and when relevant, included in the dataset.

The estimated performance of the predictors is based on the premise that the data used to train them is representative of the problem. If this is not the case, and the predictors do not perform as expected at runtime, it may be necessary to return to previous stages of the process to either generate new and more representative data (*Stage 1*) or to expand the search space of ML methods to create more accurate models (*Stage 4*). Furthermore, real systems are not stationary and often evolve, and therefore it is also possible that the underlying problem changes over time (*concept drift* [Webb et al., 2016]). There are several techniques to address this, but in the end it resumes to a lower predictive performance which may require returning to previous stages (i.e., *Stage 1* or *Stage 4*). This is in itself a complex problem for specific cases and thus falls out of the scope of this methodology and is considered for future work.

4.7 Summary

This chapter introduced an iterative multi-stage methodology that takes into consideration the particulars of using fault injection to generate failure data to develop predictive models for OFP. Each stage comprises a specific process and provides guidelines that should be considered, from generating and processing the data, to creating and deploying the predictors. Due to the exploratory nature of this process, several feedback loops are included, as it may often be necessary to return to previous stages to expand the approach. This methodology is closely related to the contributions described in *Chapter 3*, as it relies on both fault injection to generate failure data and the ML toolbox to explore the data and develop failure predictors.

Effectively implementing failure prediction solutions also requires an adequate selection of the most suitable models. It involves a strict assessment of alternative solutions using appropriate metrics, which must represent the technical needs of the target system, and their comparison using adequate datasets and procedures. As there is no accepted procedure to achieve this, existing works present several limitations, such as using ad-hoc and unreproducible approaches, or not taking into consideration the needs of the system where the predictors will operate. To address this, the following chapter introduces a conceptual framework for properly benchmarking failure prediction models, assuring a fair and sound assessment and comparison of alternative solutions. It provides detailed guidelines, from choosing the adequate metrics for assessing the performance of the predictive models taking into consideration the technical needs of the system, to comparing alternative solutions and determining the sensitiveness of the models to variations in the data.

Chapter 5

A Benchmarking Approach for ML-based OFP

Effectively implementing failure prediction involves not only extremely accurate tuning, but also an adequate selection of the most suitable model(s) for a particular system installation. More precisely, selecting a prediction model requires a rigorous assessment of alternative solutions using appropriate metrics (taking into consideration the technical needs of the target system), and their comparison using adequate datasets and procedures. This is a difficult task as the information about the performance of failure prediction models available in the literature is not sufficient to choose a predictor for a given system. To make fair comparisons, this process must be well defined, such that the assessment of the performance of the predictors provides confidence on how the results will hold in the operational scenario. Due to the lack of such a procedure, existing works use ad-hoc approaches, consider a diverse set of metrics without thorough consideration of the purpose/needs of the system, and neither statistically validate differences nor assess how sensitive the models are to the data used for training. Several other aspects should also be considered when benchmarking alternative solutions, such as validating the dataset, conducting statistical comparisons, and assessing the robustness of the models to variations in the data.

This chapter proposes a conceptual framework for benchmarking failure prediction models, assuring a fair and sound assessment and comparison, while fulfilling the following key properties [Vieira and Madeira, 2003; Gray, 1992]: *ease of use and implementation, promptness, repeatability, portability, representativeness*, and *non-intrusiveness.* It provides guidelines for implementing a procedure for benchmarking failure prediction models, including choosing the adequate metrics for the assessment depending on the application scenario, preparing and validating the workload (while leaving out of scope the dataset generation as it has already been thoroughly documented in *Chapter 3*), comparing the alternative models, and selecting the best predictor. By exploring the notion of scenarios (a realistic situation of failure prediction that depends on the criticality of the system) [Antunes and Vieira, 2015], this approach allows a better match between its outcomes and the requirements for the failure predictor operation. Three representative real-world scenarios are considered, from critical to lower-quality systems. The framework also addresses the need to assess and improve the performance of the models under small perturbations in the data (using adversarial and robustness optimization techniques), which will inevitably occur in production systems. This is of utmost importance to establish the credibility of the models obtained through the benchmarking procedure, as solutions that can only perform well on the existing data are not of interest. As this is not a typical benchmark, with specific datasets and metrics that vendors can use to sell their products, this framework is mostly for researchers and system administrators as it is intended to be used on the specific systems (and their needs/usage scenarios) where the predictors are needed. It is also possible to evolve the framework into a full-fledged benchmark (that vendors can use) by including pre-generated datasets and specific use-case scenarios, but this is left for future work.

The reasoning for proposing an approach, instead of referring to it as a benchmark, stands in the fact that a benchmark for ML algorithms typically includes a dataset against which the solutions are benchmarked. However, for OFP the dataset should represent the intrinsic characteristics of a particular system (i.e., where failure prediction will be done), thus it should be collected specifically for that system.

The architecture of the approach is illustrated in *Figure 5.1*. It is inferred from the structure of a ML approach and the organization of a dependability benchmark, enhanced with specific considerations (e.g., scenarios, robustness verification) to ensure the choice of the most adequate solution. In practice, a fair and sound assessment and comparison of failure prediction models require:

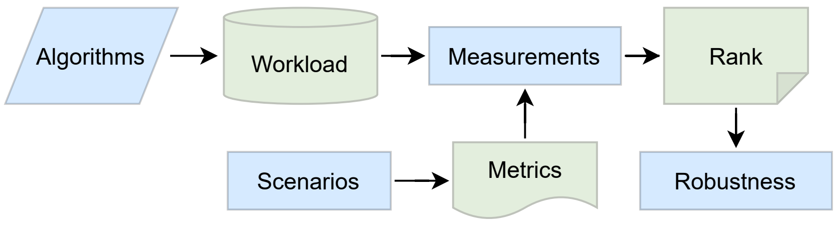


Figure 5.1: General Architecture of the Framework

- **Scenarios** – requirements representing real contexts, having constraints with different criticality, where failure prediction will be used.
- **Metrics** – allow characterizing the effectiveness of the algorithms. They must be easy to understand and allow the comparison among alternative algorithms from different perspectives and scenarios.
- **Algorithms** – a set of algorithms, techniques, and configurations that will be used against the workload and then ranked according to the chosen scenario/metrics.
- **Workload (Dataset)** – data needed to train and test the failure prediction algorithms. It should mimic the behavior of the target system, taking into account the hardware and software, workload, failure modes, etc.
- **Procedure** – rules that must be followed, including the phases that must be conducted, towards the assessment of the performance (thus generating the **Measurements** and subsequent **Rank** for the specified scenarios/metrics) and of the **Robustness** of the models to variations in the data.

Several properties were taken into consideration when designing the benchmark for the results to be sound and to minimize inaccuracies due to the procedure, namely [Vieira and Madeira, 2003; Gray, 1992]: *i*) *Ease of installation and use*: be composed of a simple executable or a document specifying how to implement it; *ii*) *Promptness*: the execution should take the shortest time possible, increasing the usability of the benchmark and of the predictors; *iii*) *Non-intrusiveness*: require minimal or no changes in the entities under analysis (i.e., the prediction models); *iv*) *Portability*: allow comparing alternative failure prediction models based on diverse approaches, scenarios, and systems; *v*) *Repeatability*: different executions must lead to the same results on a deterministic or statistical basis; and *vi*) *Representativeness*: the results must be representative of real scenarios, i.e., the prediction models must behave similarly on the target system.

5.1 Scenarios

Scenarios should be based on the technical needs and business impact of the systems in an organization. This is achieved by means of requirements in terms of the level of dependability that should be satisfied and the cost of mitigating the predicted failures before their occurrence. As an example, for a critical scenario (e.g., home banking), one wants to select a predictor with a higher detection rate, even if it raises more false-alarms than others (within some acceptable bounds), since unpredicted failures may have serious consequences. On the other hand, for a medium-quality scenario (e.g., corporate site), one may want a predictor with a high detection rate, but that does not raise too many false-alarms, since the cost of dealing with them may be high compared with the mitigation of failures.

This work considers three criticality levels representing realistic scenarios, inspired by those described by Antunes and Vieira [Antunes and Vieira, 2015]:

- **High-criticality** – failures missed may be problematic due to the criticality of the target system. However, a model that constantly raises false-alarms is also not suitable, as these will trigger countermeasures to avoid the (non-existing) failures. This way, the goal is to select a model that is able to predict the highest number of failures but also taking the number of false-alarms into some level of accountability. An example of a *high-criticality* system is a home-banking system.
- **Medium-criticality** – failures may be missed at the cost of reducing false-alarms. The goal is to select a model reporting few false-alarms at the cost of missing some failures. An example of such a system is an email server.
- **Minimum-criticality** – every false-alarm is a cause of concern due to its cost. The goal is to select the prediction model reporting the lowest number of false-alarms while still predicting some failures. An example is a forum server.

Scenarios are helpful for system administrators as they allow discerning the acceptable/expected outcomes of the failure prediction process for a system that fits in a scenario. In this framework, the user should identify the scenario(s) that match the technical needs of his system and benchmark alternative ML solutions according to its requirements (i.e., using the performance metrics that represent the goals of each scenario, as will be discussed in the next section). This allows a better match between the outcome (the predictors) and the requirements of the failure predictor operation.

5.2 Metrics

Metrics are an essential part of the benchmarking process, as they characterize the predictor and allow to fairly compare alternative models. According to Gray [Gray, 1992], benchmarking metrics should: *i*) portray the key characteristics of the entity under benchmarking, *ii*) be easy to understand and use, and *iii*) be generally accepted. Also, they must be obtained without impacting the system (i.e., the prediction models).

Revisiting some basic concepts, in a classification problem, the samples that are correctly predicted are known as *True Positives (TP)* and *True Negatives (TN)*. The positive samples (i.e., *failures*) that are predicted as negatives (i.e., *non-failures*) are *False Negatives (FN)* and the opposite are *False Positives (FP)*. Although there are several composite metrics available (e.g., precision), they should be carefully used as they are not independent of the data [Sokolova and Lapalme, 2009].

Albeit most metrics are well documented, developers are often unaware of their advantages/limitations and do not know which one to use to best fit their needs. Several metrics are included and explained in this work, leaving to the user the selection of the relevant scenario, which in turn will determine the metrics to use. In the same way that we argue that the framework should be run in the system where failure prediction is being implemented (to account for the characteristics of the system), it is also necessary that the outcome of the process fulfill the needs of the user (i.e., rank the prediction models from the most relevant perspective). For each scenario, one main metric is proposed to rank the tools (according to the goal of the scenario in the specific context of OFP) and a tiebreaker metric is used only when there is a tie (i.e., no statistical difference, as defined in the benchmarking procedure) (see *Table 5.1*).

Table 5.1: Recommend Metrics by Scenario

Scenario	Metric	Tiebreaker
High-criticality	informedness $*$ recall	recall
Medium-criticality	f-measure	precision
Minimum-criticality	markedness	precision

The next paragraphs describe the evaluation metrics and why they are adequate for each scenario, taking into account that failures are rare events (i.e., datasets are imbalanced):

- **Recall** – the proportion of failures that are correctly identified as such:

$recall = TP/(TP + FN)$. As the TN and FP are not considered and the data is highly imbalanced, this metric is used only as a tiebreaker in the *high-criticality* scenario, where the best model is the one that correctly predicts most failures.

- **Precision** – the proportion of positive predictions that were correctly classified: $precision = TP/(TP + FP)$. From a list of models predicting the same number of failures, the best is the one with fewer FPs, which is the tiebreaker adequate for both *medium-/minimum-criticality* scenarios, where the purpose is also to avoid FPs (although not the main objective).
- **F-Measure** – also known as F_1-score, the harmonic mean of precision and recall. As it gives the same importance to both precision and recall it is suitable for the *medium-criticality* scenario where it is preferable to predict fewer failures than to predict more and have an added cost due to FPs.
- **Informedness** – how consistently a predictor predicts the outcome of both a TP and a TN, that is, how informed a predictor is for the specified condition, versus chance. Every TP increases (in the proportion $1/(TP + FN)$) and every FP decreases (in the proportion $1/(TN+FP)$) the metric. In practice, due to the data imbalance, most of the time this metric prioritizes models predicting more failures, but still considering FPs, which makes it suitable for the *high-criticality* scenario.

$$Informedness = \frac{TP}{TP + FN} + \frac{TN}{FP + TN} - 1 \tag{5.1}$$

- **Markedness** – how consistently the predictor has the outcome as a marker, i.e., how marked a condition is for the specified predictor, versus chance. The metric considers the proportions of the predicted positives and negatives that are correctly classified. As failures are rare events, each FP will lower the metric more than a FN, thus satisfying the requirements for the *minimum-criticality* scenario, by minimizing the FPs whilst also classifying some failures.

$$Markedness = \frac{TP}{TP + FP} + \frac{TN}{FN + TN} - 1 \tag{5.2}$$

Something that **previous works overlooked** is that the *informedness* metric is not concerned with which condition is better predicted, that is, a model that correctly predicts 70% and 30% of negative and positive samples, respectively, has exactly the same informedness as one that predicts 30% of negative and 70% of positive samples. Logically, for the *high-criticality* scenario, the best model is the one that predicts more failures. Hence, to overcome this behavior the models are ranked based on the product of their informedness (through unity-based normalization) and recall (as shown in *Table 5.1*). This will still take advantage of the strengths of the informedness metric while assuring that the models with higher recall (among those with higher informedness) will be ranked first.

The metrics proposed attempt to address the three different common usage scenarios while dealing with the intrinsic characteristics of the dataset. Note that, time or complexity metrics are not being included. The reasoning is that, nowadays, most (trained) ML models can make nearly immediate predictions (with the exception of some 'families' of algorithms, e.g., instance-based), and thus, for the proposed approach, there should not be relevant differences between the solutions. Notwithstanding, it should be ascertained whether the selected models can make predictions within the intended frequency (e.g., one prediction per second). Concerning the training phase, although there are differences between the algorithms, this happens offline, and thus it is not as relevant compared to predictive performance. Still, as resources are finite, after ranking the various solutions based on their predictive performance, the user may take the training time into consideration. It should however be noted that different implementations of a ML technique (or even the code of the benchmark) may be more/less optimized, which will ultimately influence execution time. Instead of directly comparing absolute values the user should follow a more qualitative approach, such as defining a threshold above which the solutions are no longer acceptable. Online/incremental learning is also left out of the scope of this work, as most algorithms do not support it and it is, in itself, a complex problem. Furthermore, as mentioned before, the systems that may benefit more from OFP are large and complex servers, often subject to workloads that do not change regularly. If there are changes that require new data to be included in the models, that should lead to a new benchmarking campaign with the new data.

The user may also consider other metrics and still follow the procedure. However, they should be carefully chosen as some may be misleading in some contexts (e.g., various works wrongly use accuracy with imbalanced datasets).

5.3 Algorithms

An adequate set of algorithms, configurations, and techniques should be selected to be benchmarked. In practice, as different algorithms excel at different problems, the set of algorithms should be as diverse as possible (e.g., NN, RF). To explore the predictive potential of each algorithm the chosen set of (hyper)parameters should be representative and thoroughly selected (e.g., based on documentation, related work). Different ML techniques should be selected to handle the characteristics of the data (e.g., feature selection/extraction, sampling).

Following a thorough and detailed procedure (described in detail in *Section 5.5*), a combination of algorithms, configurations, and techniques should be used against the workload. The performance of the various solutions is then computed and ranked according to the metric of the targeted scenario. A key aspect is that the robustness of the models to minor variations in the data should be assessed and subsequently improved (if necessary).

5.4 Dataset/Workload

The dataset represents the workload that the prediction models must perform during the benchmark run.

Real workloads contain data collected from real environments. Results of benchmarks using real workloads are usually more representative. However, data from several systems may be needed, and both the portability and generalization are dependent on the systems used to collect them. *Realistic* workloads are artificial, based on a subset of operations from real systems in the intended domain. Although artificial, they still reflect real situations, are representative, and more portable than real workloads. Realistic workloads of failure data can be generated through fault injection [Durães and Madeira, 2006]. Finally, *synthetic* workloads are artificially generated based on assumptions and rules. Although they are easier to collect and provide better repeatability and portability, their representativeness is highly questionable.

Using real workloads is almost impossible in most cases, partially because failures are rare events and collecting enough data is too expensive. Thus, there have been several studies on the injection of software faults to generate realistic failure data. Techniques such as G-SWFIT [Durães and Madeira, 2006] allow the generation of large amounts of realistic failure data in a short time, thus overcoming one of the main limitations of OFP. Although this approach does not provide a specific workload (as it should mimic the system where the predictor will function), *Chapter 3* provides detailed guidelines and procedures on how to use fault injection to generate failure data for a target system. Notwithstanding, this framework includes guidelines on how the dataset should be built and validated (step *2) Dataset Building and Validation*).

5.5 Procedure

Benchmarking requires a rigorous procedure, driving the user from the assessment to the comparison of the results. The proposed procedure has five phases, depicted in *Figure 5.2*.

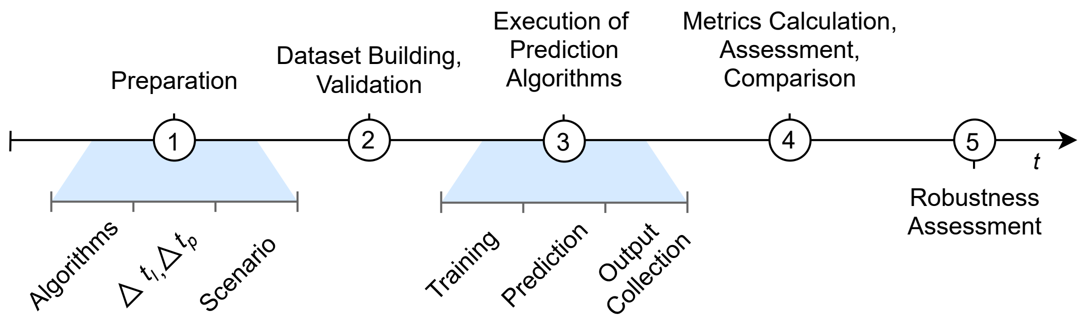


Figure 5.2: Benchmarking Procedure

5.5.1 Preparation

Preparation consists of identifying the set of parameters for benchmarking the failure prediction models (e.g., the failure modes to predict, the intervals relative to the failure prediction task; Δt_l and Δt_p). This step also encompasses the selection of the algorithms and their configurations, as well as any other techniques and configurations deemed necessary (e.g., dimensionality reduction, sliding windows). Another key aspect is to decide on the scenario to consider (*Section 5.1*) taking into account the environment in which the predictor will operate, which determines the metrics used for ranking the solutions (*Section 5.2*).

5.5.2 Dataset building and validation

Although the collection/generation of datasets is out of the scope of the benchmark (it has already been extensively overviewed in the previous sections), some rules should be taken into account in the context of benchmarking. The datasets are built from the failure data collected/generated (see *Section 5.4*) by associating information about failure/no-failure occurrence. In practice, a dataset must be a simple table where the columns are divided into a set of descriptive features (the monitored variables) and a target (whether a failure will or not occur within the specified Δt_l, Δt_p, following a procedure such as the one proposed by Salfner et al. [Salfner et al., 2010]). Each row represents an instance that contains a value for each feature and target. In the event of failure, the target should be descriptive of each failure mode (e.g., *hang*, *crash*) [Koopman et al., 1997], thus allowing the analysis of each mode separately. As these represent values taken over time, the instances should be in the same sequence as they were collected, in order to allow the evaluation of techniques that consider previous values (e.g., sliding windows). No 'false-predictors' (i.e., features directly related to the target) can exist, and the dataset should only contain sanitized data (e.g., in case of data generated through fault injection, runs that fail immediately after injection should be eliminated, as these are not representative of residual faults and would have likely been detected through traditional validation techniques).

As the dataset is provided by the user, validation is necessary to ensure that it can be used to properly benchmark different solutions. In practice, it should present different attributes, such as having a minimum number of samples for each failure mode, not having missing data, each feature should contain a single datatype, and each sample should contain the same number of features. Some algorithms can still perform when some of these attributes are not met, but this may compromise the results and limits the algorithms that can be benchmarked. Moreover, for OFP there is no reason why these attributes should not be respected.

This is probably the most demanding step, as the task of generating the datasets from the original data depends on how it was initially stored/organized. Nonetheless, this typically merely corresponds to exporting the data to the intended structure and thus can be easily automated. In the end, as long as the datasets are generated to the required structure, the proposed approach can be used on any kind of system.

5.5.3 Execution of the prediction algorithms

Each algorithm must be trained using a training dataset, while the validation and testing datasets should be used for evaluation. In this phase, the output of each predictor is collected for later processing. This phase can be divided into three parts: *i*) *training*, where the algorithm is trained using labeled data for discriminating failing from non-failing situations; *ii*) *prediction*, where each failure predictor tries to label a set of unlabeled data; and *iii*) *output collection*, where the predictions are collected. To find the best models, additional tasks may be needed such as data sampling, dimensionality reduction, and hyperparameters optimization.

5.5.4 Metrics calculation, assessment, and comparison

The predictions performed by each model are processed to calculate the metrics for the scenario being considered. Each predictor must be evaluated using the test datasets to have an estimate of the generalization error. This property (i.e., to which extent the results will hold in the operational scenario) is closely related to the confidence one may have in the benchmarking results. As discussed in *Section 2.2* and *Chapter 4*, over the years various methods have been proposed to assess the performance of ML models (e.g., *k-fold cross-validation*). Several techniques have also been developed to account for the sequential and autocorrelation of the data in a time series (e.g., *out-of-sample* [Tashman, 2000]). However, given the experiment-based nature of OFP (e.g., there is no sequence between multiple failures, and failures will always be at the end of the experiments) such techniques are not adequate.

As previously argued (and as will be shown in *Chapter 6*), using *k*-fold cross-validation for developing OFP solutions leads to overfit models that cannot generalize properly (i.e., when the best models developed using *k*-fold cross-validation, which supposedly could correctly predict practically all non-failure and failure samples, were put online they failed to detect almost all the incoming failures). To avoid this, and to obtain a more realistic estimate, the performance of the models should be assessed using ELOOCV (introduced in *Chapter 4*), a novel validation approach for OFP based on the leave-one-out cross-validation. In short, ELOOCV considers every experiment except one for training and then tests on the held-out experiment, iterating over all the experiments. A simple illustration of the process was shown in *Figure 4.3*. This approach mimics an online environment, where the models will make predictions on completely unseen experiments.

As many algorithms (e.g., NN, K-means) do not perform adequately when dealing with features that have different scales (which often occurs in datasets for OFP as the features monitor different parts of the system) the data should be normalized (e.g., using *normalization*, scaling the values to a fixed range [0, 1], or *standardization*, also called *Z-score normalization*, which scales the data to have $\mu = 0$ and $\sigma = 1$) [Irrera and Vieira, 2014]. Given the lack of boundaries (and possible outliers) that are common in OFP, the use of z-score normalization is advisable. Each experiment should be run multiple times with different seeds for statistical support.

The user should analyze the results of various models and select the one that best fits his requirements, depending on the selected scenario. Statistical tests are required to guarantee that the best solutions are in fact statistically different. Briefly, *paired* statistical tests should be used (as each experiment uses the same data) and the *normality* and *homogeneity* of the variance should be assessed to decide between *parametric* and *non-parametric* tests [Field, 2013]. When there are no significant differences, the ranking should consider the tiebreaker metric. The user must also define how many solutions are of interest (i.e., the top N, on which the statistical comparisons will be done) and how many additional models should be analyzed after N if no significant differences are found.

Based on the ranking of the solutions, the user may also take into consideration other aspects, such as the interpretability of the models or the time they take to train (as discussed in *Section 5.2*). Moreover, the user may also use non-subjective analysis of benchmarking results (e.g., using Analytic Hierarchy Process (AHP), a multicriteria decision-making technique that allows mathematically expressing subjective and personal preferences [Martínez et al., 2014]).

5.5.5 Robustness assessment

The performance of the models should be assessed using data with small variations (as will likely occur in production). This step is essential in the benchmarking procedure to assure that the models will also be able to perform accordingly when dealing with new unseen (and inherently different) data. The first step is to craft new samples to assess the robustness of the models. These samples may be obtained through simplistic approaches (e.g., random/guided perturbations) or crafted adversarial samples. In general, there are three common approaches to generate adversarial samples: *white-box* (uses knowledge regarding the configurations and architecture of the models), *gray-box* (which uses only partial knowledge of the system), and *black-box* (relies only on the outputs given by the models). Ultimately, all approaches try to identify the minimal perturbation that leads the models to a wrong output.

To retain the characteristics of the original samples the perturbations must be constrained. As discussed in *Section 2.2*, adversarial perturbations are typically measured using L_p-norms. Due to the nature of a complex system (where multiple variables may vary), L_∞ is the most adequate norm, as it only measures the largest feature variation. Contrary to generating adversarial samples on images, where a human can typically validate if a perturbed image is still similar to the original, it is much harder to assess the imperceptibility of the changes in tabular data, which are often comprised of hundreds of numeric continuous features (an issue thoroughly discussed by Ballet et al. [Ballet et al., 2019]). Given that this work recommends using Z-score normalization, which transforms the data to have unity standard deviation, a maximum perturbation of 0.3 was identified based on informal experimentation (i.e., by studying and assessing the magnitude of the perturbations on the original values of the features) as being adequate to insert variations without compromising the characteristics/label of the original sample.

As an example, which will also be discussed in the experimental evaluation in *Chapter 6*, for a maximum L_∞ perturbation of 0.3 the *average cpu percentage of system apps* feature, whose values range from 0% to 88%, would change at most by 0.81%. Regarding the *average apps memory use (MiB)*, which ranges from 32MB to 505MB, it would mean a maximum variation of 5.12MB. While dependent on how the values of the features are distributed, with Z-score standardization a perturbation of 0.3 is a conservative value regarding expected variations in the data. This value, which can also be defined on a per-feature basis, is ultimately context-dependent. Another characteristic that must also be considered when generating adversarial samples is that different features will likely have different hard bounds (e.g., cpu frequency cannot be lower than 0) which cannot be violated.

The next step is to use defensive techniques to improve the robustness of the models against such variations (if necessary). Although there are several techniques available [Biggio and Roli, 2018], two approaches are worth mentioning. The first is to use *data augmentation/adversarial training* techniques, which consist of augmenting the training data to create more robust models. While effective, these are heuristic defenses, without convergence and robustness guarantees [Biggio and Roli, 2018]. The second is based on *robust optimization* techniques, where adversarial learning is formulated as a minimax problem [Biggio and Roli, 2018]. Although this is more efficient and provides formal guarantees, such works are often algorithm-specific and may not be available or be easy to apply to some of the algorithms considered in the benchmarking process.

5.6 Summary

Effectively implementing failure prediction is a complex task. It requires extensive and accurate tuning and a rigorous assessment of alternative solutions using appropriate metrics, followed by a thorough comparison considering adequate datasets and procedures. To provide confidence that the results obtained will hold on an operational scenario this process must be well defined and strictly followed. Notwithstanding, as such a process is not available, existing works rely on ad-hoc approaches, considering a limited set of methods and a diverse set of metrics, having little consideration for the comparison and sensitiveness of the models to variations in the data.

This chapter introduced a conceptual framework for benchmarking predictive models for OFP, assuring a fair and sound assessment and comparison of alternative solutions. It provides a detailed procedure comprised of several steps and considerations, such as choosing the most adequate performance metrics taking into account the needs of the system, preparing and validating the workload, assessing and comparing alternative failure predictors, and ultimately selecting the one that best fits the operational requirements. It explores the notion of scenarios, which are realistic situations that depend on the criticality of the system, providing a better match between its outcomes and the requirements of the failure predictor operation. The framework also attempts to address a common issue in benchmarking: how to assure that the results of the process will hold when using different data. This is achieved by exploring the use of adversarial ML to assess and improve the performance of the models to minor variations in the data. This proposal is the last item of the framework described in *Chapter 1* and builds on top of all the previous contributions, from the techniques and artifacts proposed in *Chapter 3* to the detailed methodology introduced in *Chapter 4*.

To demonstrate how these contributions can be used to develop accurate failure predictors on a new and recent complex system, the next chapter presents a comprehensive experimental evaluation targeting the Linux OS. It comprises an extensive fault injection campaign, considering different workloads representing different usage scenarios, several fault types, and various failure modes. The generated data are then thoroughly explored and an extensive study is conducted on the use of ML techniques, from traditional to state-of-the-art algorithms, and their applicability to OFP. The resulting models are then compared, exploring how different operational scenarios determine which solution is best to satisfy its operational requirements. Each section provides a detailed analysis of the implementation, results, and insights of each contribution.

Chapter 6

Experimental Evaluation

The contributions described in the previous chapters provide theoretical knowledge, guidelines, and procedures to overcome the most relevant challenges to the use of OFP. This chapter presents an extensive experimental evaluation to demonstrate how they can be used in practice.

This chapter can be divided into three parts: *generating failure data*, *developing failure predictors*, and *benchmarking alternative predictive solutions*. As this is a fairly extensive experimental evaluation, we suggest that, if the reader is interested in the process of generating the data, then focus on *Section 6.1*. On the other hand, if the main interest is in creating and developing predictive models using ML techniques, this is thoroughly detailed and analyzed in *Section 6.2* and *Section 6.3*. Finally, if the reader is mostly interested in how to properly compare and benchmark alternative predictive solutions, this is developed in *Section 6.4*.

6.1 Generating Failure Data

The first step of any failure prediction effort is to generate/collect the data to support the development of predictive models for OFP from the specific system where these will be deployed. This section presents the details and overviews the process of using the techniques proposed in *Chapter 3*, discussing the specifics of configuring and deploying a testbed (following the guidelines proposed in *Section 3.1*) and presenting a comprehensive fault injection campaign (conducted following the process described in *Section 3.2*). A thorough analysis on the impact of running experiments simultaneously is done to assess the effectiveness and isolation provided by the testbed. The generated data are then explored from the perspectives of the impact of the faults, of the distribution of failures per fault type, as well as of the most relevant characteristics that may influence the process of developing predictive models.

6.1.1 Experimental Setup

The experiments were conducted on a PowerEdge R630 with 2 Intel®Xeon®E5/2650 CPUs (with a NUMA node each, hence the testbed had to take this into account). The machine has 64GB of DDR-4 and 2 Samsung 850 Pro 512GB SSDs.

Linux was selected as the target OS for this experimental evaluation. It is often the chosen platform for research and it can also be found in the most varied applications, from small embedded to large enterprise-grade systems. For containment/flexibility, the QEMU (with KVM) hypervisor was used. The guest OS is an up-to-date 32-bit Long-term Support (LTS) Linux kernel 3.16.82. This version was selected due to the intended fault injector dependencies. The fault injector used in these experiments [Yoshimura et al., 2012] (which is further discussed in the next sections) was only functional up to kernel version 3.7 but we ported it to the latest LTS version available of the same major version, kernel version 3.16.82. The host runs 64-bit Ubuntu 18.04.02 with the 5.0.0 low-latency kernel. Single-core VMs with 2GB of memory were used. The system was configured (using *cpusets*) to execute sixteen experiments simultaneously (as managing the virtual machines and keeping data in memory also consumes resources, 16 simultaneous experiments was the maximum possible in this setup to avoid resource starvation and swap memory use).

The experimental process was automated using Python and Pexpect [Spurrier, nd]. As mentioned before, although different sources of data can be used for OFP (e.g., system logs), this study focuses on monitoring the system state through its metrics (e.g., CPU, memory). Thus, several system metrics (all numeric) were collected from the VMs every second using Netdata [Netdata, nd] (923 metrics, which can be seen in *Appendix B*).

Automating interactions with complex programs such as VMs is a complicated task by default. On top of that, as fault injection often leads to the corruption/malfunction of the target system, the testbed must be resilient and handle errors in the interactions by design. Whenever possible, the various processes are asynchronous (e.g., monitoring, failure detectors) to avoid halting the execution when the target system begins to misbehave. All the considerations/optimizations described in *Section 3.1* were implemented to increase the performance isolation of the experiments. More precisely: *isolcpus* and *cset-shield/set*, NUMA and cpu affinity, low-latency kernel, preallocating memory, hugepages, limiting CPU frequency on the isolated cores, real-time scheduler, avoiding IRQ and RCU callbacks on isolated cores, and cleaning dirty blocks on disk after each batch of experiments. At the end of the experiment, all resources are cleared/freed and all execution is gracefully terminated. The testbed also monitors the state of resources on the host to avoid overloading its execution (i.e., when executing thousands of experiments it is easy to reach system limits, e.g., open file descriptors) and inadvertently influence the experiments.

6.1.2 Workloads

A workload is needed to exercise the system and study the effectiveness of the testbed and the impact of the injected faults. As the workload influences the behavior of the system (with and without injected faults) it must be selected considering the technical needs of that same system. For this study, *stress-ng* [Ubuntu, ndf] was used to generate the workloads. *stress-ng* was selected as it contains various stressors designed to exercise several physical subsystems, as well as various kernel interfaces. This allows simulating realistic workloads, such as cpu-intensive (e.g., computation) and memory-intensive (e.g., video-editing) tasks.

For assessing the isolation provided by the testbed, all stressors available in stress-ng were considered (i.e., *cpu, io, matrix, stream, zlib, hdd, memtrash*, and *vm*), which can be seen in *Table 6.1*. Due to the large number of experiments that are required to achieve statistical relevance, for the fault injection campaign we selected three of the workloads, which provide common usage scenarios: *io* (an I/O-intensive workload, committing buffer cache to disk), *cpu* (a CPU-intensive workload), and *matrix* (a memory/cache and floating-point-intensive workload). The workloads executed for 10 minutes (after fault injection). As a thorough fault injection campaign requires executing a very large set of experiments, a trade-off had to be defined between the duration of the workload and the chance of activating the fault and subsequent failure. Based on an exploratory evaluation, and as will be shown in *Section 6.1.6*, in this setup most of the failures occurred relatively close to the fault injection and thus 10 minutes is an acceptable duration to allow the system to stabilize and provide enough time for faults to be activated and lead to failures. Nevertheless, some failures may take longer to manifest and therefore require longer experimental campaigns, which will be explored in future work.

Table 6.1: Workloads - stress-ng [Ubuntu, ndf]

Workload	**Description**
cpu	exercise the CPU using several methods (e.g., computing Ackerman functions, complex floating point operations)
io	continuously calling sync(2) to commit buffer cache to disk
matrix	perform various matrix operations on floating point values
stream	exercise memory bandwidth, loosely based on the STREAM benchmarking tool [McCalpin, 1995]
zlib	compress and decompress random data using zlib, exercises CPU, cache, and memory
hdd	continuously writing, reading, and removing temporary files
memthrash	thrash and exercise a 16MB buffer in various ways to trip thermal overrun
vm	continuously calling mmap(2)/munmap(2) and writing to the allocated memory

6.1.3 Assessing Isolation

Sixteen experiments were conducted sequentially (to establish a baseline) and sixteen were executed in parallel for each stressor, for both the *isolated* environment (i.e., where the various techniques to achieve isolation were implemented) and *non-isolated* environment (where no specific isolation techniques were implemented).

To assess the potential influence of other tasks running on the host, a third set of sixteen experiments executing simultaneously was conducted for each stressor and environment while a cpu and memory-intensive workload was running on a separate session. A summary of the configurations can be seen in *Table 6.2.*

Table 6.2: Isolation Experiments

Workloads	*cpu, io, matrix, stream, zlib, hdd, memthrash, vm*
Environments	**isolated** (isolation techniques are implemented), **non-isolated** (no isolation techniques are implemented)
Executions	**sequential** (the experiments are executed sequentially), **parallel** (the experiments are executed simultaneously, i.e., at the same time), **parallel with host load** (the experiments are executed simultaneously while a cpu- and memory-intensive workload is running on the host)
Experiments	16 experiments per set (i.e., for each combination of workload, environment, and execution), a total of 768 experiments

A summary of the results can be seen in *Table 6.3.* For each environment (i.e., *non-isolated* and *isolated*) the first group (i.e., *cpu*, *io*, *matrix*, *stream*, *zlib*) are those where isolation is noticeable and the second (i.e., *hdd*, *memtrash*, *vm*) where it is not. The metric used for comparison is *bogo operations* (a throughput measure). Although it is not an accurate benchmarking metric, it allows comparing the performance across different environments [Ubuntu, ndf].

As can be observed, without isolation all stressors have considerable variation between sequential and parallelized execution, which is further aggravated when the host is executing other tasks (e.g., *cpu* stressor presents a variation of approximately 20%, executing 88860, 81347, and 70664 operations, respectively). Furthermore, without isolation standard deviations are systematically higher (e.g., the *io* stressor had a standard deviation of 187552 for the non-isolated and only 3363 for the isolated environment). On the other hand, when isolating the experiments it is possible to observe that all the stressors in the first group (e.g., *cpu* to *zlib*) have similar results across the various environments, regardless of other tasks executing on the host (e.g., *cpu* stressor executed 38594, 38527, and 38370 operations for the *sequential*, *parallel*, and *parallel with host load* environments, a maximum difference of 0.5%).

As expected, benchmarks that stress components that cannot be parallelized are affected by running experiments simultaneously (e.g., *hdd* stressor operations dropped 38%, from 169910 to 105453). Also, when comparing the performance of sequential experiments between isolated and non-isolated environments, it is possible to observe that there is a considerable difference. This is due to limiting the CPU frequency, e.g., when run sequentially, the *cpu* stressor executed on average 88860 operations on the non-isolated environment against 38594 on the isolated.

Each set of 16 experiments took on average approximately **20 minutes** when run

Table 6.3: Experiments Isolation

Non-Isolated

Benchmark	**Sequential**	**Parallel**	**Parallel w/ Host Load**
cpu	88860 (173)	81347 (314)	70664 (10456)
io	1620905 (10787)	1375507 (187552)	1373051 (82853)
matrix	1781923 (2976)	1461701 (292359)	1338686 (257073)
stream	542 (0)	513 (10)	510 (4)
zlib	17372 (228)	15596 (1073)	14414 (1248)
hdd	185388 (28279)	103555 (696)	103191 (789)
memthrash	37486 (300)	25686 (3598)	19719 (2135)
vm	4894297 (213710)	3827312 (665271)	2065351 (593175)

Isolated

Benchmark	**Sequential**	**Parallel**	**Parallel w/ Host Load**
cpu	38594 (73)	38527 (65)	38370 (96)
io	738995 (2020)	734746 (3362)	730612 (3828)
matrix	775231 (688)	774583 (367)	771314 (1782)
stream	465 (0)	461 (1)	460 (1)
zlib	7518 (250)	7630 (20)	7521 (171)
hdd	169910 (19139)	105453 (7892)	103176 (796)
memthrash	30673 (218)	25121 (344)	23562 (1634)
vm	31599440 (530104)	2511423 (207344)	2230932 (78575)

Legend: {operations average} ({standard deviation})

simultaneously, while the sequential approach took approximately **3 hours and 45 minutes**. Scaling this to a larger experimental set translates to a considerable difference in execution time (e.g., an initial fault injection campaign of 2176 experiments took approximately **46 hours**, while executing them sequentially would have taken **22 days**).

Using the techniques proposed in *Section 3.1*, it was possible to create an automated testbed that could take advantage of the system resources through simultaneous executions with minimal interference (except for workloads that depend on components that cannot be parallelized). Fixing the CPU frequency to its minimum also led to lower individual throughput compared to the non-isolated environment. Notwithstanding, it is possible to profile the CPU frequency under the target workload to minimize the performance gap while avoiding reaching thermal limits (i.e., identifying the maximum frequency that does not overheat the CPU). On the other hand, without isolation, the results were considerably influenced by other experiments/tasks running on the system.

6.1.4 Fault Injector and Fault Models

A comprehensive fault injection campaign was conducted (following the process described in *Section 3.2*) to generate the failure data on the deployed testbed. Briefly revisiting some concepts, a fault injection campaign typically requires executing a large number of experiments with and without fault injection for a given workload (or set of workloads). Runs in which faults are injected are known as *fault injection runs*, which can be either *failing runs* (if a failure occurs) or *non-failing runs* (otherwise). Additionally, *golden runs* (i.e., runs where no faults are injected) are also necessary to understand the behavior of the system in the absence of faults. A fault injection environment is usually comprised of various components [Hsueh et al., 1997], as illustrated in *Figure 2.5*: *i)* a *controller* (which controls the experiment); *ii)* a *fault injector*; *iii)* a *fault library/model*; *iv)* a *monitoring system*; and *v)* a *workload* to exercise the system.

For these experiments, we used an updated version [Yoshimura et al., 2012] of a well-known SWIFI fault injector [Ng and Chen, 1999], widely used in various previous works [Swift et al., 2006; Depoutovitch and Stumm, 2010; Kwon et al., 2016; Cotroneo et al., 2018]. It uses object-code modification to inject bugs into the kernel of a running OS. Instead of injecting multiple faults randomly (a common/default approach), the kernel execution was profiled under the various workloads (i.e., *cpu, matrix*, and *io*) and faults were only injected within the kernel functions executed by the workload to improve the chance of triggering a fault. As faults were being injected in functions with a high probability of being executed, a single fault was injected per run.

The injected faults range from low-level (e.g., bit-flips) to high-level (e.g., memory allocation) faults [Ng and Chen, 1999]. The latter are the most relevant and intend to approximate the assembly-level manifestation of real C-level programming errors. The injector disassembles the binary of a randomly selected function in the kernel text segment. Since the faults are context-dependent, it analyzes the disassembled code and searches for proper locations in which each type of fault can be injected [Yoshimura et al., 2013], as illustrated in *Figure 2.6.* The injected faults emulate various types of real faults: *assignment faults*, *control faults*, *parameter faults*, *omission faults*, and *pointer faults* [Yoshimura et al., 2012; Cotroneo et al., 2015]. In total, 16 fault types representing the most common types of faults committed by programmers (which can be seen in *Table 6.4* [Yoshimura et al., 2012]) were considered. It should be noted that not all injected faults cause faulty behaviors (e.g., bugs inserted on a rarely/never executed path/condition will rarely/never produce an error). As the execution of the system is not entirely deterministic, three separate experiments were conducted for each fault location and workload.

Table 6.4: Fault Types

Fault Type	Description
ALLOC	kmalloc returns NULL
BCOPY	makes string functions overrun
BRANCH	deletes branches
DSTSRC	destroys assignments
FREE	deletes kfree
INIT	omits initialization
INTERFACE	omits function arguments
INVERSE	flips predictions
LOOP	destroys loop condition
IRQ	deletes restoration of interrupts
NULL	omits NULL check
OFF-BY-ONE	e.g., ja change into jae
PTR	destroys pointers
SIZE	makes heap alloc. smaller
TEXT	bit-flip in the kernel text segment
VAR	allocates huge local valuable

6.1.5 Failure Modes

Various failure modes were monitored during the experiments, described in *Table 6.5*: *crash* (OS crashes), *hang* (OS hangs), *performance* (the performance deviates more than 5%, to tolerate statistical variances, than the lowest baseline value), *infinite execution* (workload has not finished after 15 minutes), *I/O* (no longer writes to disk), and *filesystem corruption* (using the *fsck* functionality). The system logs were also monitored for non-fail-stop failures (e.g., *unable to handle kernel null pointer*) which were grouped into different 'failure categories' (considering their probable cause): *cpu/execution-related*, *memory-related*, and *kernel-related* failures. If failure messages of different categories are detected (nearly) simultaneously their combination was considered and the most probable cause selected (e.g., if *Kernel BUG at [...]*, a 'generic' message that was associated with *kernel-related failures*, was followed by an *invalid opcode: [...] SMP*, which we associated with *cpu/execution-related failures*, the latter would take precedence as it is a more specific/probable failure/cause). The complete list of failure detectors considered in the experimental process can be seen in *Appendix D*.

Although these failure modes are partially based on the C.R.A.S.H. scale [Koopman et al., 1997] (e.g., *crash* maps to *Catastrophic*; *hang* maps to *Restart*; *corruption* and *infinite execution* map to *Silent*) it was decided to consider and analyze the various failures separately to preserve the granularity and create more accurate/precise models.

Table 6.5: Failure Modes

	Failure Mode	**Description**
fail-stop	crash	OS becomes corrupted and the system crashes or reboots
	hang	OS becomes unresponsive and must be forcefully terminated
non-fail-stop	memory	memory-related failures detected in logs (e.g., *segmentation fault*)
	cpu/exec	cpu/execution-related failures detected in logs (e.g., *invalid opcode*)
	kernel	kernel-related failures detected in logs (e.g., *kernel bug at [...]*)
post-hoc	performance	the performance deviates more than 5% than the baseline
	corruption	the filesystem becomes corrupted
	infinite-exec.	the workload takes 50% more time than expected
	I/O	the system can no longer write to disk

6.1.6 Discussion

A large number of experiments were executed, with, and without, fault injection, as can be seen in *Table 6.6*. The total number of fault injection experiments was 4472, 4417, and 4483, for workloads *cpu, matrix,* and *io*, respectively. Additionally, 100 runs without injecting faults were conducted for each workload to establish a baseline. To gather as much information as possible, the experiments were only terminated if a fail-stop failure occurred or more than 30 non-fail-stop failures were detected (such failures are no longer relevant for these studies because the goal is to predict the first failure, as afterward the system is already impaired). As a result, it is possible to have experiments with multiple failures.

Table 6.6: Experiments per Workload

Workload	**Fault Injection**	**Golden**
cpu	4472	100
matrix	4417	100
io	4483	100

On top of the fact that only a small fraction of the experiments led to the observation of failures, not all of those were suitable for OFP. Runs in which the failure occurred immediately after fault injection (or activation, monitored using *kprobes* [Keniston et al., nd]) were considered *invalid* as they are not representative of residual faults (i.e., such faults would have been detected by traditional validation techniques), and would not provide enough data points to create predictive models. Furthermore, there were some experiments where it was not possible to monitor the fault activation (i.e., it was not possible to insert probes due to the specific location of some faults), and also some rare occurrences of faults that compromised the system monitor (i.e., it was not possible to monitor the state of the system at every second). Finally, some failures were observed only after the workload ended (labeled as *late*).

Table 6.7 shows a summary of the fault injection, including the number of fail-stop failures observed during the experiments. As can be seen by comparing the total number of non-failing experiments (the first column) with the experiments that led to failures (i.e., columns *Invalid, Failure Late, Failure without Activation*, and *Corrupted Failure*), only a fraction of the fault injection experiments actually led to failure. Moreover, most of those failures were too close to fault activation (i.e., *invalid* experiments, which can be seen in column *Invalid*) and thus cannot be used for OFP. It is also possible to observe that out of the 16 fault types some rarely caused failures (e.g., *FREE*), while some others almost always led to *invalid* failures. In practice, only a subset (e.g., *INVERSE*, *LOOP*) actually led to useful failures. Overall, from the 13372 experiments conducted (excluding golden runs), 78.2% of the runs did not lead to the observation of any failure; 18.6% were considered *invalid*; 1.7% led to failures being observed/logged after the workload ended; 0.1% had valid failures but incomplete monitoring; in 0.2% it was not possible to register the fault activation timestamp; and **1.2%** led to valid failures that could be used for OFP.

The fail-stop failures were also analyzed to get some insight into how different types of faults lead to the most severe failures. As it is possible to observe in the last 4 columns of *Table 6.7*, *crash* failures were considerably more common than *hang* or *infinite execution*. Additionally, some fault types were particularly effective at crashing the system (e.g., *DSTSRC*, *PTR*, and *TEXT*).

The experiments per fault type depend on the number of viable fault candidate locations (i.e., locations where a fault of a given type can be injected) in the code executed by each workload. Hence, the discrepancies in the number of experiments per fault type are explained by some types having considerably more candidate locations. One of the reasons why there are so many invalid experiments is because several non-fail-stop failures are considered (parsed from the system logs, e.g., *unable to handle kernel null pointer*), which often occurred immediately after injecting a fault, despite sometimes leading to other failures later in the execution. Although the number of invalid experiments would be considerably lower if only fail-stop failures were considered, such faults/experiments are not representative because they would have been detected by traditional validation techniques.

To have a better understanding of when the failures occurred, the Time-to-Failure (TTF) (i.e., the time between fault activation and failure) of the relevant experiments (i.e., those that were valid and occurred during the execution of the workload) was analyzed. *Figure 6.1* illustrates the TTF of the *cpu* workload (other stressors presented similar distribution). As it is possible to observe, most failures

Table 6.7: SWIFI Summary

								fail-stop			
Fault \ **Result**	**Non-Failure**	**Invalid**	**Failure Late**	**Fail. w/o Actv.**	**Corrupt Failure**	**Failure**	**Total**	**Excess. Fail.**	**Infin. Exec.**	**Hang**	**Crash**
ALLOC	195	75	3	0	0	0	273	15	0	0	0
BCOPY	143	27	1	0	0	0	171	4	0	0	11
BRANCH	1282	248	29	2	0	**14**	1575	69	27	0	41
DSTSRC	1153	325	30	5	0	**18**	1533	70	17	5	89
FREE	411	0	5	0	0	0	416	0	0	0	0
INIT	1019	148	20	5	0	**10**	1202	66	2	7	20
INTERFACE	490	166	24	3	0	**22**	705	52	22	10	13
INVERSE	1021	250	46	0	5	**22**	1344	59	19	1	67
LOOP	1236	168	33	3	6	**22**	1468	28	13	7	28
IRQ	171	49	0	0	0	**8**	228	9	3	9	0
NULL	237	6	0	0	0	0	243	0	0	0	0
OFFBYONE	1189	34	12	0	0	**7**	1242	9	2	0	9
PTR	977	330	15	0	0	**37**	1359	107	18	13	98
SIZE	36	0	0	0	0	0	36	0	0	0	0
TEXT	752	663	10	2	0	0	1427	121	18	24	192
VAR	149	0	1	0	0	0	150	0	0	0	0
Total	10463	2489	229	20	11	160	**13372**	609	141	76	568
Percent.	78.2%	18.6%	1.7%	0.2%	0.1%	1.2%	100%				

occurred nearly 50/100 seconds after the activation. Additionally, there were also two other clusters, near 250 and 550 seconds. This suggests that most of the faults cause failures in a relatively short term after activation while others can take a much longer time to manifest (and may eventually be harder to predict as the symptoms will likely be spread over time).

To build the dataset used to develop failure predictors, we selected the baseline runs and failures that met the following criteria: *i)* the fault activation was observed; *ii)* the failure did not occur immediately after fault activation; *iii)* the failure occurred within the workload execution time; and *iv)* the system metrics were consistently collected throughout the experiment. Although failures that occurred after the end of the workload can be used for other studies, this work is only concerned with predicting failures that occur during the execution of the business processes (i.e., workload) and not during the cooldown/shutdown phase.

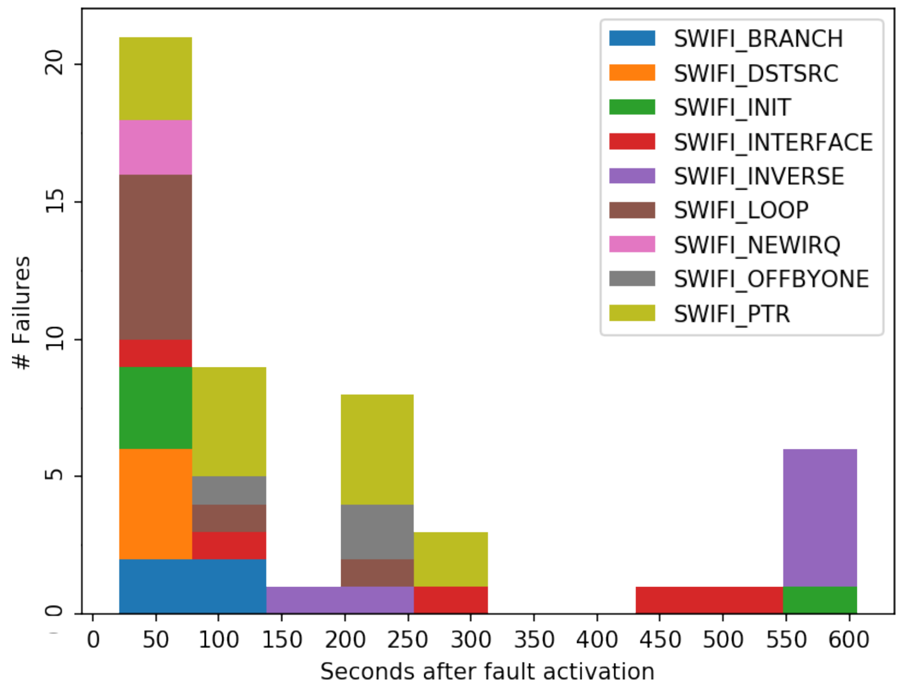


Figure 6.1: *cpu* Workload Time-to-Failure (TTF)

In practice, failures for which it is not possible to assign a specific timestamp for their occurrence were excluded as this is required for accurately labeling the data samples (otherwise samples may be incorrectly labeled as failures; e.g., *performance* failures are logged at the end of the workload, but currently it is not known when the misbehavior began). Runs where the fault activation could not be monitored were also excluded as it is not possible to assure there was a separation between fault activation and the failure event.

When exporting the Netdata data, all constant metrics were removed (i.e., metrics that contained a single value throughout all the experiments), as these do not add any value but may increase complexity (e.g., the pipes/sockets of the DHCP process). Additionally, although Netdata monitors hundreds of metrics, some are transient (e.g., metrics related to email servers, which are typically not exercised in the selected workloads). Thus, only those that were present throughout all the experiments for the duration of the workload were selected. As all metrics returned by Netdata were stored (and not just a predefined subset), another concern was that they could contain 'false-predictors/data-leaks', that is, features that contain information directly related to what will be predicted (i.e., failure/no-failure; e.g., as the experiments have a fixed workflow, metrics such as *system uptime* may provide information related to the failures that has no meaning on a real system). Hence, all potential false-predictors were removed. The final dataset has 371 features (which can be seen in *Appendix C*) out of 923 metrics collected by Netdata (which can be seen in *Appendix B*).

6.2 Creating Failure Predictors

The main goal of OFP is to be able to detect incoming failures in the near future based on past data and the current state of the system. This allows taking preemptive measures in an attempt to avoid or mitigate their consequences. The premise is that besides causing a failure, an error (which is an incorrect state caused by a fault) may cause the system to behave erratically (also known as *symptoms*) [Salfner et al., 2010]. Thus, ML algorithms are the most logical choice to predict failures based on *symptoms monitoring*, as they can detect complex patterns in the data that would not likely be found otherwise. Revisiting some concepts introduced in *Section 2.1*, OFP is a decision process that tries to predict, at a time t, whether a failure is going to occur within a precise time, called *lead-time* Δt_l. These predictions are valid for a given time window, called *prediction-window* Δt_p. Thus, at time t, the model should predict if there will be a failure in the interval $[t+\Delta t_l, t+\Delta t_l+\Delta t_p]$ [Salfner et al., 2010], as illustrated in *Figure 2.4*. Transposing OFP terminology to ML, the monitored system metrics are known as *features*, and the values of those metrics/features at a given time t constitute a *sample*.

After generating the data (see the previous section), the next step is to explore them and conduct an initial study to have a better understanding of how different ML techniques can model the problem. This also provides insights into which techniques are more promising and potentially discard others. This section presents an initial data-driven study on using ML algorithms and techniques to create failure predictors for OFP. The results and influence of the different techniques are carefully analyzed, and comparisons are made with some of the most used algorithms for OFP (e.g., SVM).

6.2.1 Data Preparation

Due to the fact that most features in the dataset are based on different scales, a Z-score standardization was applied to the data. To be able to study in-depth the influence of the OFP parameters (i.e., lead-time and prediction-window), the combinations resulting from $\Delta t_l = [20, 30, 40]$ and $\Delta t_p = [20, 40, 80, 120, 160, 200]$ (as also described in *Table 6.8*) were considered. The samples were labeled following the approach proposed by Salfner et al. [Salfner et al., 2010] and illustrated in *Figure 4.2*, generating different datasets for each pair of $\Delta t_l, \Delta t_p$, i.e., 18 for each workload. A minimum lead-time of 20 seconds was considered as the shortest window to take preemptive measures. The maximum was limited to 40 seconds because greater values would result in removing too many experiments (due to having TTFs lower than the lead-time) and ultimately some failure modes would no longer be represented.

As most fault injection runs did not result in a failure being observed, to make the experiments feasible, all the datasets were limited to a maximum of 100k samples (which is large enough to include all failure runs and several non-failing and golden runs). The only exception was for the SVM algorithm (its complexity increases considerably with the size of the data) for which the datasets were limited to 50k. As it is likely that different types of failure will require distinct preemptive measures, the classes of the problem (those to be predicted) correspond to the different high-level failure modes defined in *Section 6.1*: *control* (i.e., non-failure), *hang*, *crash*, *cpu/exec* (i.e., CPU/execution-related failures), *memory* (i.e., memory-related failures), and *kernel* (i.e., kernel-related failures).

A descriptive and exploratory analysis was conducted to have a better understanding of the data. While it is not possible to include here every analysis, one of the most interesting insights was that the averages of the values of the features per failure mode (for each workload and $\Delta t_l, \Delta t_p$ pair) are noticeable different, thus suggesting that they may be distinguishable, as depicted in the example in *Figure 6.2*. As can be seen, there is a considerable difference between the samples from the various failure modes and the non-failure samples (they are not noticeable in the graph but they are mostly around 0). Additionally, some failure modes appear to have very distinguishable traits (e.g., *cpu/exec*), while others may not be so perceivable (e.g., *kernel*).

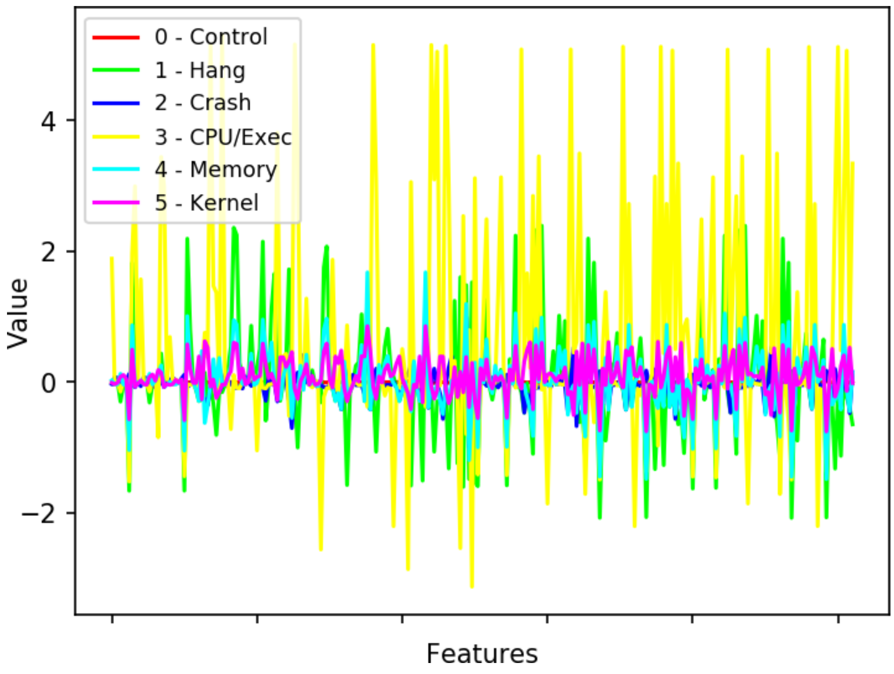


Figure 6.2: *cpu* Workload [20, 40] Features Means

The distribution of the classes in the datasets can be seen in *Figure 6.3* (by workload and $\Delta t_l, \Delta t_p$). For readability purposes, the non-failure samples were excluded, which were always around 97k. It should be noted that for each run only the data up to the first failure is considered (i.e., if it is a non-fail-stop failure, subsequent data/failures are discarded), as the goal is only to predict the first system failure. As can be observed in the figure, *cpu/exec* and *kernel* failure samples are, in general, the most represented classes. *Crash* samples are also common and constant across the different workloads, while *memory* less so. *Hang* failure samples are the least represented class and are not even present for the *io* workload. It is also possible to observe a decrease in the number of samples when increasing lead-time, which is due to some experiments being excluded because their TTF is lower than lead-time. On the other hand, when increasing the prediction-window, the number of samples increases, as more samples are considered as indicative of (future) failure occurrence. Comparing the number of failure samples (which ranged from ~1k to ~4k for the $\Delta t_l, \Delta t_p$ pairs [60, 40] and [20, 160]) with the number of non-failure samples (~97k) it is possible to conclude that this is a highly imbalanced dataset (as expected), which requires some care concerning the techniques and metrics to be used.

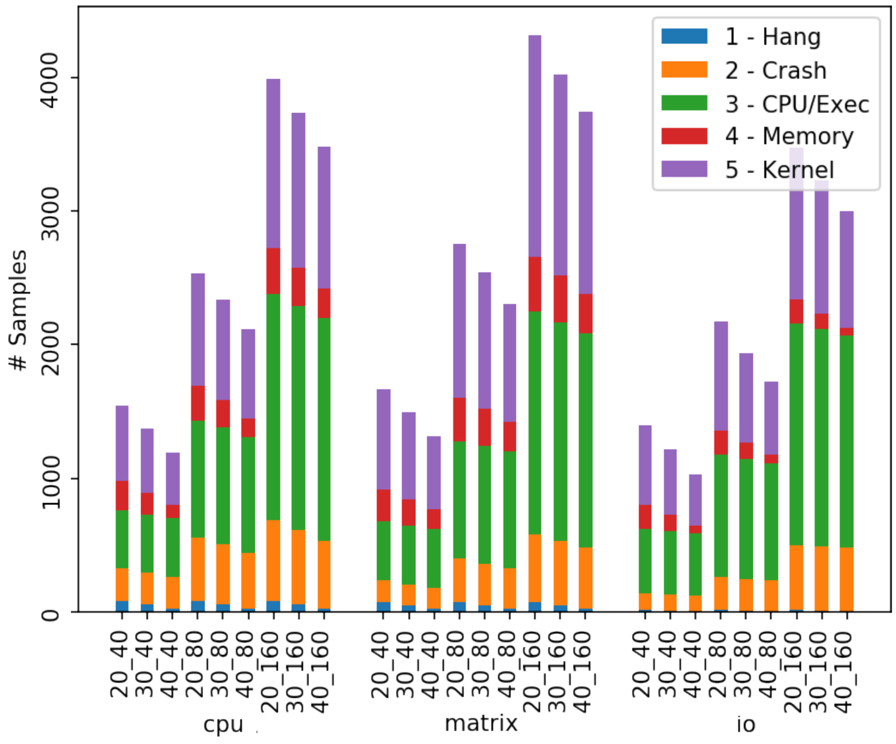


Figure 6.3: Classes Distribution

Last but not least, heat maps were used to observe how correlated the features were, which easily allows identifying strong (both positive and negative) correlations, as illustrated in *Figure 6.4*. This allowed observing that there are some highly correlated features, especially between related metrics, such as *system cpu percentage* and (overall) *cpu percentage*. Some 'unrelated' metrics are also highly correlated (e.g., *kernel* and *softirq CPU percentage* have a 76% correlation) and thus can likely be removed without the loss of significant data. Notwithstanding, most metrics are not highly correlated and therefore their removal may lead to loss of information.

6.2.2 ML Algorithms and Techniques

To understand how different algorithms model the problem at hand, a comprehensive/diverse set was selected, as can be seen in *Table 6.8*. Given the imbalanced nature of the dataset, the influence of different sampling techniques was also studied. As the dataset also contains a considerable number of features (i.e., 371) two common feature selection methods were used: *variance* (which removes features with 0 variance) and *correlation* (which removes highly correlated features, >90%). The list of methods used can be seen in *Table 6.8*. To fine-tune the mod-

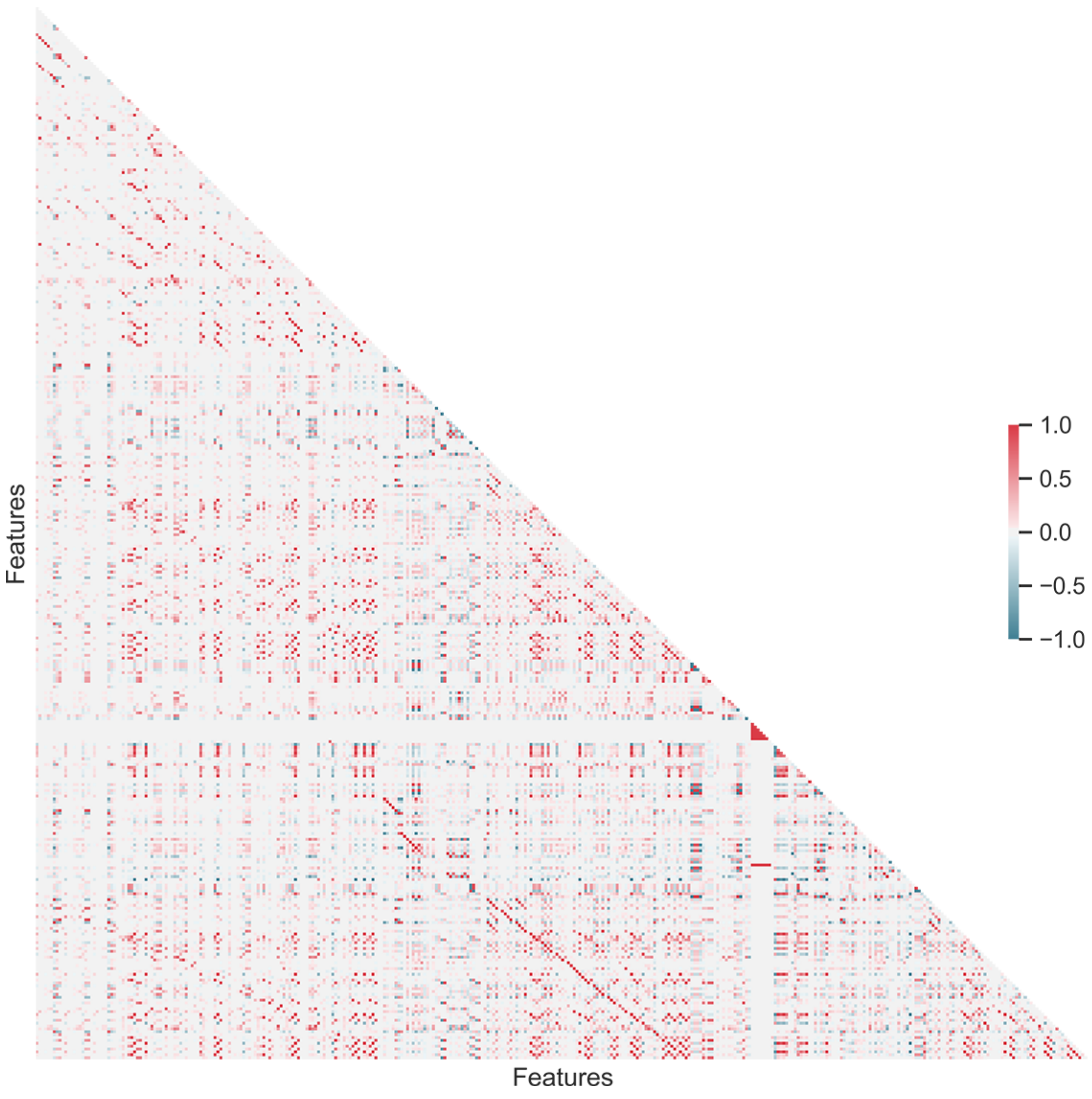


Figure 6.4: CPU workload [20, 40] Heat Map

els, a grid-search approach was applied considering the hyperparameters defined in *Table 6.9*. These include the most relevant parameters for each algorithm and the values were selected based on their respective documentation (e.g., [Pedregosa et al., 2011; Chen and Guestrin, 2016]) to provide a good representation of their predictive performance.

To estimate the performance of the models, a 5-fold *stratified cross-validation* was applied and each experiment was run 5 times with different random seeds to reduce possible bias. As we argue in *Chapter 4* that cross-validation is not adequate for OFP, some clarification is warranted. OFP is a specific case of a time series, and it does not intend to forecast the values of some variable, instead, it attempts to predict whether the current state of the system is (or not) indicative of an event (that will occur in the future). Moreover, failures occur at the end of the experiments execution, and there is no relation between experiments (i.e., no 'past' or 'future' after each failure). To complicate things further, validation techniques typically used for time series are not applicable

Table 6.8: Techniques Used in the Experiments

Process / Step	**Techniques**
Workloads	stress-ng: cpu, matrix, io
$\Delta t_l / \Delta t_p$	[20, 30, 40] / [20, 40, 80, 120, 160, 200]
Feature Selection	Variance, Correlation
Sampling	Random under/oversampling
Algorithms	SVM, NN (Multilayer Perceptron (MLP)), DT, Bagging, RF, XGBoost,

Table 6.9: Algorithms' Hyperparameters

Alg.	**Hyperparameters**
SVM	**kernel**: [linear, RBF, poly], **C**: [.01, .1, 1], **gamma**: [.1, scale], **degree**: [2, 3, 4]
NN	**hidden_layers**: [(100,1), (100, 5, 1)], **activation**: [logistic, relu], **solver**: [lbfgs, adam]
DT	**min_samp_split**: [001, .01, 2], **max_feat.**: [.1, .55, None], **min_samp_leaf**: [.001, .01, 1]
RF	**estimators**: 100, **max_feat.**: [.1, .55, auto], **min_samp_leaf**: [.001, .01, 1], **min_samp_split**: [.001, .01, 2]
Bag.	**max_features**: [.1, .55, 1.0], **estimators**: 100
XGBs.	**estimators**: 100, **learning rate**: [0.5, .3, .1], **gamma**: [0, 0.1, 0.3], **subsample**: [0.5, 0.7, 1]

for OFP (e.g., *rolling-origin*). Ultimately, OFP falls in a gray area, as no clear guidelines exist. Nevertheless, because ultimately OFP is a supervised learning problem, in these initial analyses we used the standard ML approach for such problems, i.e., *stratified k-fold cross-validation.* This decision is further supported by the fact that related works on OFP also used cross-validation (e.g., [Irrera and Vieira, 2014]) and that some authors argue that cross-validation can still be used for time series problems [Bergmeir et al., 2018]. However, as will be shown in *Section 6.3*, for OFP this leads to overoptimistic results that do not hold in production.

To compare and rank the different models it is necessary to select a performance metric that allows characterizing their effectiveness. Several metrics are available, but they should be carefully used as they are not independent from the data [Sokolova and Lapalme, 2009]. In fact, while it is common to see widespread use of metrics such as *accuracy* and *precision*, their use should be thoroughly considered. Concerning accuracy, it is not an adequate metric for imbalanced data because it does not take into account the representation differences between classes. In fact, in a highly imbalanced dataset, a model that correctly predicts all negative samples and no positive samples (i.e., just predict everything as negative, no failure will ever occur) would just have slightly lower accuracy than a model that could also predict all failures (i.e., a 'perfect' model). Concerning precision, while relevant, alone it is also not adequate, as, for example, a model that predicts everything as negative with the exception of a single correctly predicted failure (i.e., 1 true-positive) would have a precision of 100%, with a (near) 0% recall. Additionally, the metric should also take into consideration the requirements of the system where the predictor will operate. In fact, as argued in *Chapter 5*, the definition of the 'best' model is not straightforward. As an example, for a more critical context (e.g., home banking), one wants to select a predictor with a higher detection rate, even if it raises more false-alarms than others (within some acceptable bounds), since unpredicted failures may have serious consequences. On the other hand, for a medium-quality context (e.g., corporate site), one may want a predictor with a high detection rate, but that does not raise too many false-alarms, since the cost of pro-actively dealing with those false-alarms may be high compared with the mitigation of failures.

In the OFP context, a common/general goal is to predict as many failures as possible but also taking into account the number of false-positives (i.e., a system that is constantly giving false alarms is not very useful for many scenarios). Thus, considering this context, the F_2-score metric was used in this analysis, which gives double the importance of recall compared to precision (i.e., it is more important to predict failures, recall, yet a compromise must be made with the number of false-positives, precision). Additionally, the analysis and assessment of the various models was based on confusion-matrices, as they allow a clear identification of how many samples of each class are being (in)correctly predicted, also providing valuable insights regarding which classes are being confused with which.

The Propheticus framework, introduced in *Section 3.3*, was used to conduct these experiments as it facilitates exploring the problem/data and easily testing and assessing ML methods.

6.2.3 Binary Classification

The first step was to assess the simplest approach to OFP: predict incoming failures regardless of their type (i.e., a 'generic' failure). For the various workloads, the best SVM model was able to correctly classify nearly every non-failure sample, as can be seen in *Figure 6.5* (obtained with a Radial Basis Function (RBF) kernel for the *cpu* workload and $\Delta t_l, \Delta t_p$ pair [40, 40]). However, SVM models could barely predict any failures. On the other hand, for the *cpu* workload and $\Delta t_l, \Delta t_p$, NN models were not only able to correctly classify almost every non-failure sample, but also to correctly predict nearly 80% of the failure cases.

Regarding the DT algorithm, it performs considerably better than the previous ones (the results for the *cpu* workload and $\Delta t_l, \Delta t_p$ pair [20, 40] can be seen in *Figure 6.6*). Its models could distinguish between non-failure and failure samples quite accurately, correctly predicting up to 99.9% and 94.5% of non-failure and failure samples respectively.

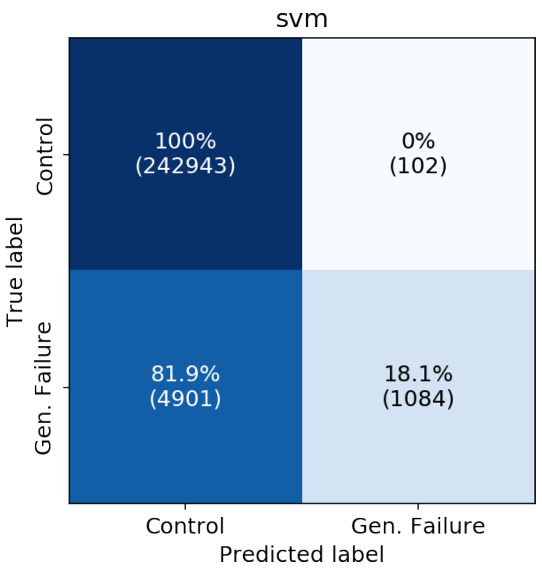


Figure 6.5: SVM w/ RBF, *cpu*, [40, 40]

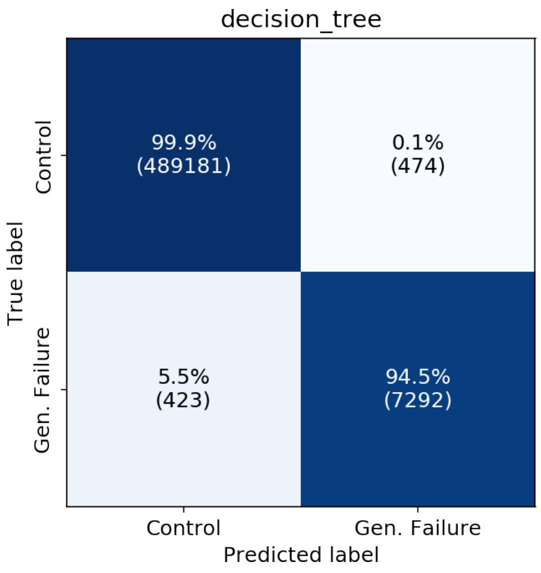


Figure 6.6: DT, *cpu*, [20, 40]

NOTE: the numbers displayed in the confusion-matrices pertain to the total number of samples per class (~100k/50k total) multiplied by the number of executions (5)

As for the ensemble methods, for the *cpu* workload and $\Delta t_l, \Delta t_p$, RF correctly classified 99.9% of non-failure and 94.2% of failure samples while Bagging managed to predict 97.2% of the failure samples. XGBoost performed even better, correctly classifying 99.9% of non-failure and 97.8% of failure samples. These results suggest that DTs can model the problem better than SVMs and NNs (although simple, over the years DTs have shown very good results in several problems). Concerning Bagging, RF, and XGBoost, they are all ensemble methods that leverage several individual learners (e.g., mitigating bias/variance) and often achieve better performance.

6.2.4 Classification by Failure Mode

After establishing that it is possible to create accurate predictive models in this new problem/dataset, the next step was to study if they could predict different failure modes (i.e., multi-class prediction). Besides providing more information, this can be useful to allow taking failure-specific preemptive measures.

Although not particularly surprising (given the poor results in the binary classification task), SVM models still could not accurately predict failures. As an example, while the best model for the *cpu* workload and $\Delta t_l, \Delta t_p$ pair [40, 40] could acceptably predict *hang* and *cpu/exec* failures (98.1% and 91.8% respectively), all the remaining failure modes had low performances (~50%). Concerning NNs, also for the *cpu* workload and $\Delta t_l, \Delta t_p$ pair [40, 40], the best model could predict almost every non-failure sample (99.8%) and had an acceptable performance on *hang* and *cpu/exec*, predicting 93.3% and 81.7% respectively. Still, it was only able to predict 70.9%, 43.2%, and 58.9% of *crash*, *memory*, and *kernel* failure samples, respectively.

DT models for the *cpu* workload and $\Delta t_l, \Delta t_p$ pair [20, 40], can not only distinguish non-failure from failure samples but also correctly differentiate between almost every failure mode, with minimal confusion/misclassifications (as can be seen in *Figure 6.7*). Bagging and RF had similar and slightly better performances, and XGBoost was the best, correctly predicting most failure samples (as depicted in *Figure 6.8*).

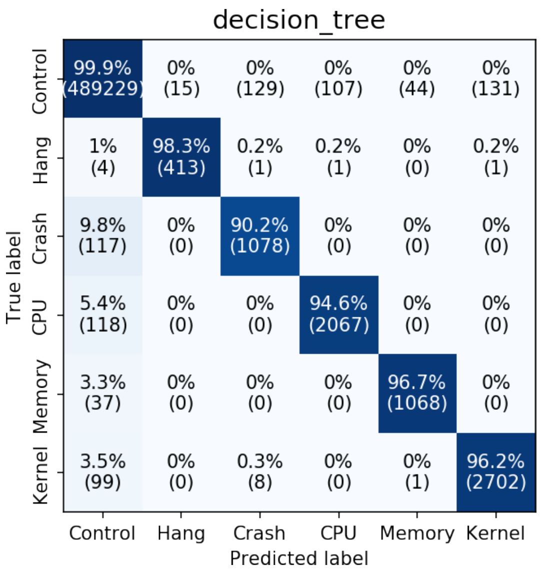


Figure 6.7: DT, *cpu*, [20, 40]

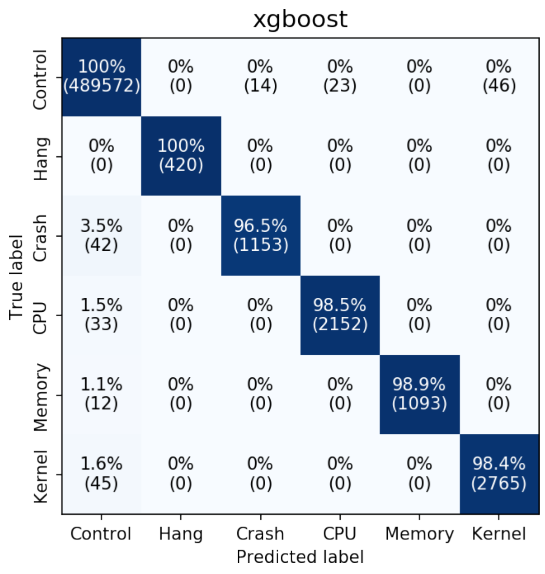


Figure 6.8: XGBoost, *cpu*, [20, 40]

To assess the influence of different sampling techniques, the experiments were executed also using random over/undersampling. While undersampling led to models that could predict more failure samples (e.g., XGBoost reached nearly 99% for most failure modes), this came at the expense of considerably more false-positives (~5%). Oversampling often resulted in improved performance in some classes (i.e., for the same configurations used in *Figure 6.7*, the performance of *cpu/exec, memory,* and *kernel* increased by ~0.2%) at the expense of others (i.e., *hang* and *crash* dropped by ~2%). Similarly, feature selection typically lowered the performance of most algorithms (e.g., for the same configurations used in *Figure 6.7* the performance for the *memory* and *kernel* failure modes dropped to 95.6% and 94.8%, respectively) although it sometimes also led to small improvements (e.g., the performance on *crash* samples increased to 91.1%).

To study/assess the influence of the workloads and how similar/predictable were the failures observed, the data from the experiments of the workloads were combined. Due to the systematic lower performance of SVM and NN from now on the results will focus on the remaining, more promising, algorithms (i.e., DT, Bagging, RF, and XGBoost).

While some algorithms suffered a small performance loss for some failure modes (compared to the individual performance on some workloads), it increased for others. For the DT algorithm, which can be seen in *Figure 6.9*, it is possible to observe that (compared to the *cpu* workload in *Figure 6.7*) while it could predict more *crash* samples (from 90.2% to 93.2%) other failures had a small loss (e.g., *memory*, from 96.7% to 94%). For other algorithms, such as the XGBoost algorithm, the performance often improved, as can be seen in *Figure 6.10*.

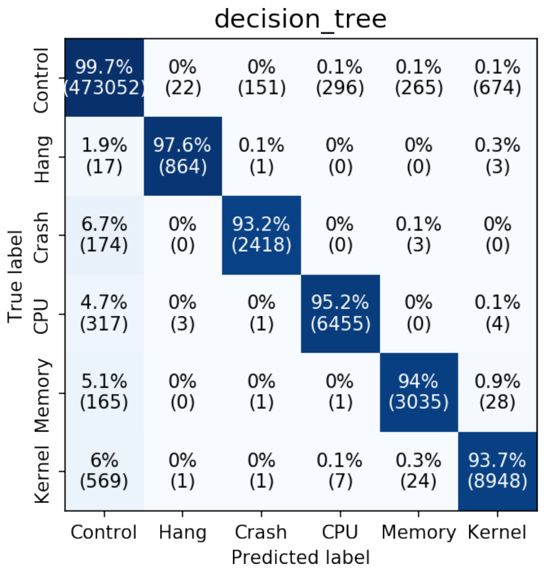


Figure 6.9: DT, Comb., [20, 40]

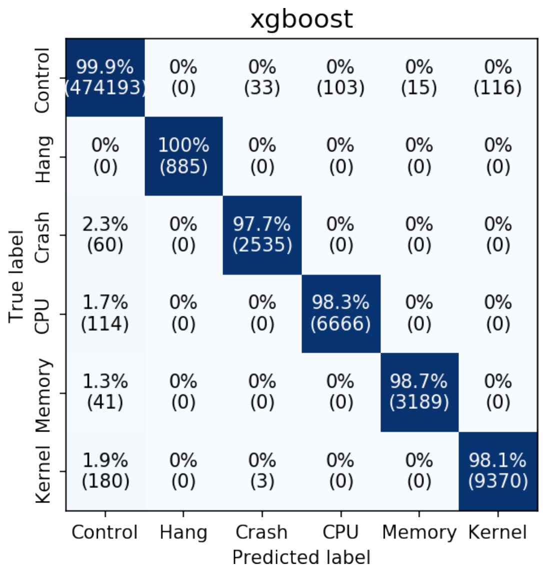


Figure 6.10: XGBoost, Comb., [20, 40]

To understand which features are more relevant, we briefly assessed which features were considered more important to the various algorithms. While it was expected to have a subset of features that exhibited higher importance, the fact is that no feature/subset had a particularly high relevance. Among those deemed more important, there are several pertaining to system/kernel/processes RAM/memory (e.g., committed, SLAB, buffers), I/O activity (e.g., logical writes), and CPU/-processes information (e.g., number, pipes/sockets, forks), which are common to most models. Nonetheless, different algorithms gave different importance to the same features.

As shown above, it is possible to assert that some algorithms can create models that accurately predict and distinguish between failure modes, even when considering data from multiple workloads. Still, the influence of the OFP parameters (i.e., lead-time and prediction-window) on their performance should also be studied. *Table 6.10* presents the performances of the best models obtained for the $\Delta t_l, \Delta t_p$ pairs considered ([20, 40], [30, 40], and [40, 40]). The performance varies with the different $\Delta t_l, \Delta t_p$ pairs and, in general, most algorithms lose performance as lead-time increases. It is also possible to observe that, overall, both the SVM and NN models have considerably lower performance compared to the remaining algorithms. Although DT performs considerably well, it was outperformed by RF and Bagging. XGBoost systematically outperformed the remaining algorithms.

To further explore the relation between lead-time and prediction-window, the best DT, Bagging, and XGBoost models were selected and trained/tested with the $\Delta t_l, \Delta t_p$ pairs described in *Table 6.8*. The results are depicted in *Figure 6.11* that shows that the performance of the models tends to improve with greater

Table 6.10: F_2-score per lead-time (40 sec. prediction-window)

	[20, 40]			[30, 40]			[40, 40]		
	F_2	**Prec.**	**Rec.**	F_2	**Prec.**	**Rec.**	F_2	**Prec.**	**Rec.**
SVM	.386	.675	.357	.391	.646	.360	.393	.639	.362
NN	.851	.893	.842	.815	.867	.804	.783	.840	.770
DT	.943	.937	.944	.935	.931	.936	.905	.907	.904
RF	.964	.986	.958	.963	.989	.957	.929	.986	.917
Bag.	.970	.987	.966	.970	.989	.966	.939	.985	.929
XGB	**.985**	.989	.984	**.983**	.991	.981	**.978**	.989	.976

prediction-windows, stabilizing after a certain size (e.g., ~60 for XGBoost). Regarding lead-time, every algorithm performed better with smaller distances, and the performance of both DT and Bagging dropped considerably with the largest lead-time. The best model can be seen in *Figure 6.12*, which correctly predicts almost every sample.

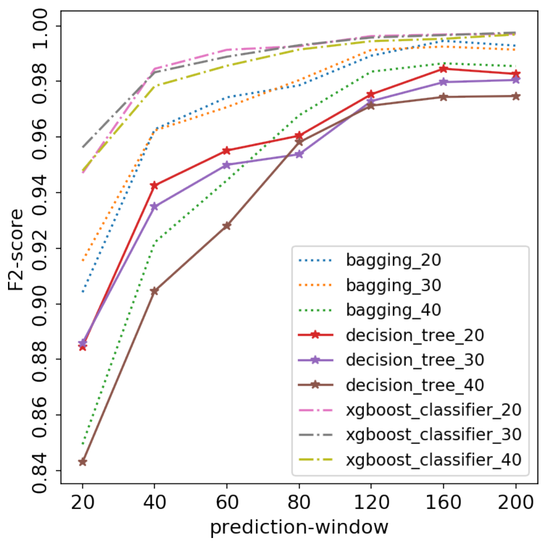


Figure 6.11: F_2-score and $\Delta t_l, \Delta t_p$

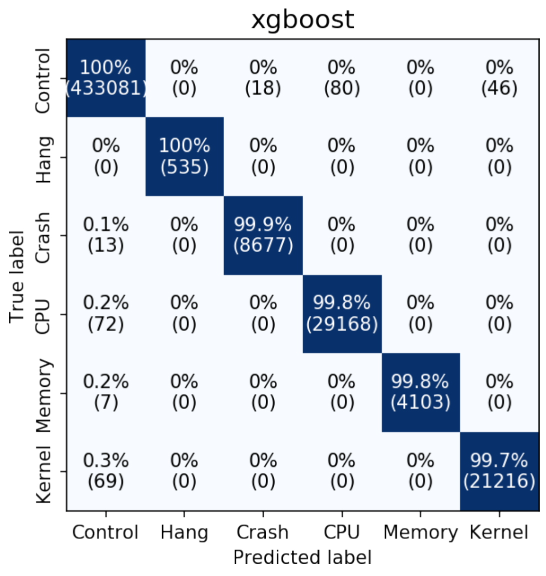


Figure 6.12: XGBoost, Best, [30, 200]

A last analysis is to study if including the data from previous timestamps leads to better models. For this, the *sliding-window* technique was used, which codes the previous values as extra features in the data. As the number of features is multiplied by the number of seconds considered in the sliding-window and the dataset already contains many features, this analysis focuses on the best algorithm and the most promising $\Delta t_l, \Delta t_p$ pairs previously found (i.e., XGBoost and [20, 60], [20, 120], [20, 200]). The sizes of the sliding-windows considered were [1, 2, 4].

The results can be seen in *Table 6.11*. Because the performance obtained by XGBoost was already very high, the sliding-window did not lead to any improvement.

For the DT algorithm, which still had some room for improvement, there were also no significant gains and no obvious differences between different sliding-window dimensions. Notwithstanding, in general, it led to a small increase (~2%) in the ability to predict *crash* and *memory* failures (and sometimes even *hang*) at the expense of slightly lower performance on the remaining failure modes.

Table 6.11: F_2-score per sliding-window

		F_2			Precision			Recall		
		1	**2**	**4**	**1**	**2**	**4**	**1**	**2**	**4**
[20, 60]	**XG**	.991	.990	.990	.993	.993	.993	.991	.990	.990
	DT	.955	.947	.943	.951	.944	.939	.956	.948	.944
[20, 120]	**XG**	.996	.996	.995	.997	.997	.997	.996	.995	.994
	DT	.975	.967	.965	.975	.965	.964	.976	.968	.966
[20, 200]	**XG**	.997	.997	.997	.997	.997	.997	.997	.997	.997
	DT	.983	.977	.977	.981	.976	.976	.983	.978	.977

6.2.5 Discussion

Similar to existing related works (e.g., [Zhang et al., 2020; Irrera et al., 2013b]), some algorithms were able to accurately distinguish between non-failure and failure samples. Although several algorithms were studied (e.g., SVM, NN), tree-based models systematically performed better. Besides their increased performance (for this problem), such models can, if needed, be easily interpreted. An example of a DT model for the *memory* failure mode can be seen in *Figure 6.13*. Although this was not thoroughly explored in this work, it allows (to some extent) understanding which features and values are used to distinguish between the different classes (e.g., if the metric *mem_kernel_MiB_average{chart=mem.kernel,family=kernel,dimension=VmallocUsed}* is lower or equal to 0.699 it is a non-failure sample, otherwise, it will transverse down the tree analyzing the values of different metrics). Ultimately, this may provide relevant insights, such as which components of the system have an abnormal behavior prior to a failure.

Although our study started by simply trying to predict failures, regardless of their type, the ability to predict different failures modes can be useful (e.g., such predictions can be used to take failure-specific preemptive measures). Our results show that not only it is possible to predict incoming failures, it is also possible to accurately distinguish between different types of failure. Additionally, the performance observed is similar for the different workloads (and for their combination) which, contrary to previous work [Irrera and Vieira, 2014], suggests that the failure symptoms are, to some extent, generalizable.

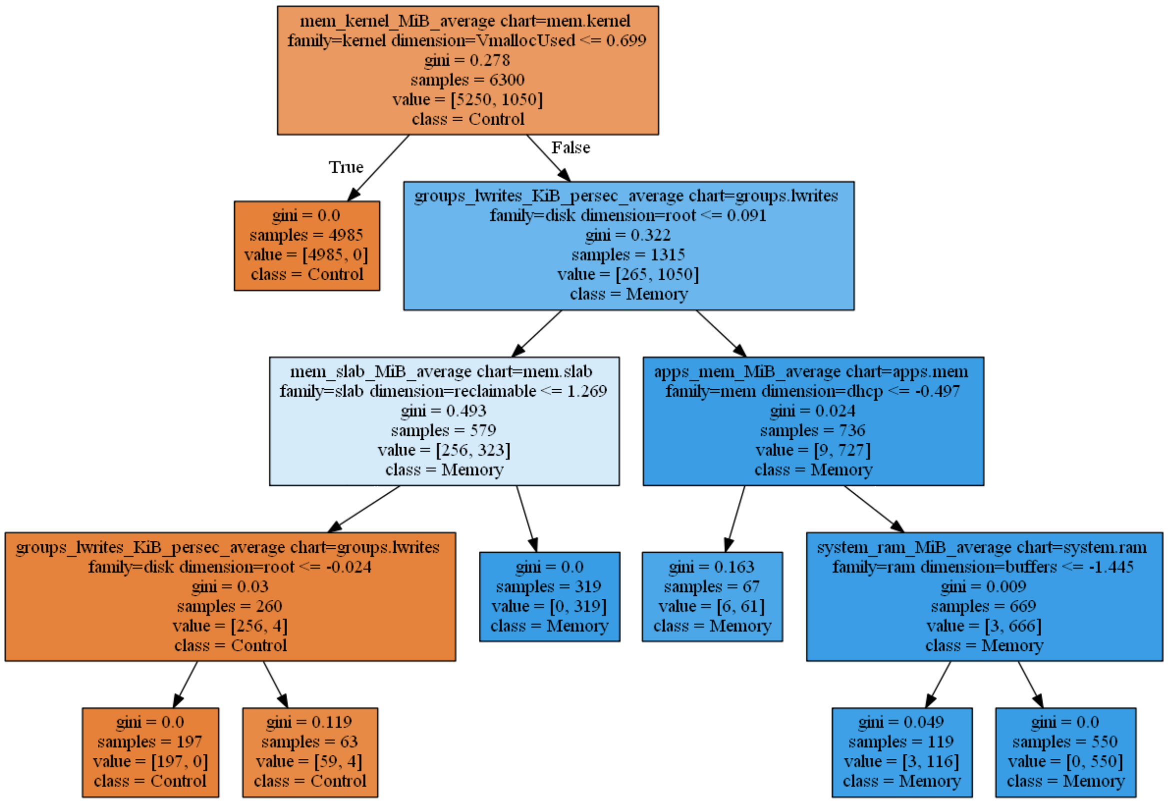


Figure 6.13: Decision Tree Model for Memory Failures

Removing highly correlated features generally led to lower performance. One interpretation is that, although correlated, they all have relevant information. Using undersampling led to models that could better predict failures, at the expense of more false-positives. While this may fit some scenarios, given the imbalance in the data even a small increase in false-positives may be an issue. Oversampling often led to small improvements in predicting some failure modes at the expense of others. Such trade-off could eventually be used to improve the prediction of more severe failures by forfeiting on less severe ones.

Results also show that the OFP parameters exert a considerable influence. As the *lead-time* increases, the performance of the models drops (which can be easily understood as it is making more distant predictions). Still, lead-time does not need to be greater than the time the system takes to react [Salfner et al., 2010]. However, the performance of the models improves as the *prediction-window* increases. A larger prediction-window means that a longer timeframe of samples is considered as indicative of an incoming failure. One interpretation could be that the time relation between symptoms and failures is not constant, and a larger prediction-window allows some flexibility. Still, from a certain point onward, the performance stabilizes (as can be seen in *Figure 6.11*), and will likely start losing performance because samples without symptoms (i.e., samples where there is no out-of-norm behavior) will be labeled as predictors of failure (because the prediction-window becomes too broad). This suggests that the ideal $\Delta t_l, \Delta t_p$ values may vary across the failure modes (which is in fact the case, as will be shown in Section 6.3).

Taking into consideration the values of the features in previous instants (i.e., *sliding-window*) did not lead to a systematic increase in performance. Notwithstanding, the ability of some algorithms to predict some failure modes increased, suggesting that there may be some relationships that can be leveraged. Still, the best solutions were achieved without sliding-windows, indicating that the characterization of the system given by the features is distinguishable enough.

The fact that no specific subset of features presented particularly high importance suggests that the system state (and failure-prone/symptoms) is not characterized by the misbehavior of a particular subsystem, but rather by the system as a whole. Nonetheless, while indicative, the importance of the features varies with different methods [Strobl et al., 2007; Parr et al., nd] and requires a more thorough analysis to make further conclusions.

To validate if the previous observations could be generalized, an exploratory study was also conducted on an existing failure dataset targeting Windows XP. The goal of this analysis is to provide assurances and external validity concerning the ability to create accurate failure predictors regardless of the fault injector or OS. The details of this evaluation can be found in *Appendix A*. In general, results show that using the proposed techniques it is also possible to develop accurate failure predictors for an entirely different complex system. Once again, the study highlights the need to consider a comprehensive set of algorithms and techniques that take the characteristics of the problem into consideration. Tree-based algorithms were again able to achieve better results for the different failure modes, and the results suggest that failures are similar across different environments (i.e., bare-metal and virtualized). This analysis also explored the creation of heterogeneous ensembles, that leverage the bias of different ML algorithms. The combination of models created using different techniques appears to produce ensembles where they complement each other in a constructive way, thus considerably improving the overall performance. Heterogeneous ensembles may be an interesting direction when individual learners are not enough.

6.3 Developing Predictive Models

The techniques and validation methods used in the previous sections are the standard approaches in ML. However, both fault injection and OFP present certain characteristics that render such techniques inadequate. As an example, when the models obtained in the previous section (which were supposedly very accurate) were used on new unseen experiments, they could not predict most of the failures (as shown in *Figure 6.15* and briefly discussed in *Section 6.3.4*, most of the failure samples were predicted as non-failure).

This section presents an instantiation of the methodology proposed in *Chapter 4*. It iterates through the various stages and analyzes how such a process compares and leads to better and more realistic estimates than classic/naive approaches. As will be shown in detail later in this section, this also allowed identifying and solving some issues with the previously defined taxonomy. Using the methodology proposed in *Chapter 4* it was possible to create accurate predictors that were able to detect almost every failure without raising false-alerts on baseline/golden runs. Several ML methods (e.g., Bagging, SMOTE) and OFP parameters (e.g., $\Delta t_l, \Delta t_p$) were considered to create failure-specific predictive models. These experiments were all conducted using the Propheticus tool with a custom plugin to implement ELOOCV, the novel experiment-wise leave-one-out cross-validation proposed in *Chapter 4*.

6.3.1 Generate Failure Data

The data generated in *Section 6.1* were used in these experiments and thus this section just briefly recalls the most relevant details. The dataset was generated using fault injection on a Linux kernel 3.16.82 (following the guidelines proposed in *Section 3.1 and Section 3.2*). System metrics (all numeric) were collected every second using Netdata [Netdata, nd], which can be seen in *Appendix B* (the complete set) and *Appendix C* (the subset of metrics used to develop the predictive models). The faults were injected with an updated version of a well-known fault injector [Yoshimura et al., 2012], which uses object-code modification to inject the faults into the kernel of a running OS. The injected faults range from *low-level* (e.g., bit-flips) to *high-level* (e.g., memory allocation). As many real-world scenarios and applications are cpu-intensive (e.g., computation, video processing), this analysis focuses on the *cpu* workload. The dataset contains a total of 4472 fault injection experiments and 100 runs (*golden runs*) were conducted to establish a baseline behavior.

Various failure modes were monitored: *crash* (OS crashes), *hang* (OS hangs), *performance* (the performance of the workload deviates more than 5% than the baseline), *infinite execution* (workload has not finished in 15 minutes), and *filesystem corruption* (using the *fsck* functionality). Other non-fail-stop failures were also monitored through the system logs (e.g., *unable to handle kernel null pointer*). To detect them and accurately register their timestamp, several failure detectors were deployed, which continuously monitor the state of the system (e.g., a *hang* failure is detected when the machine stops executing multiple tasks, such as ping, I/O, socket connections). The complete list can be seen in *Appendix D*.

6.3.2 Process, Cleanse, Augment Data

Not all experiments that lead to failure are suitable for OFP. As this was already analyzed in *Section 6.1*, this section just briefly overviews the most relevant aspects. Experiments where the failure occurred immediately after the fault injection/activation, or where it was not possible to monitor the fault activation, were excluded. Failures that occurred after the workload ended were also not considered, as this study is concerned with predicting failures during the execution of the business process. For the remaining experiments, the collected metrics were analyzed, which showed that, in some cases, the monitoring system was corrupted and thus the corresponding experiments could not be used for OFP. For the *cpu* workload, only ~0.9% (40) of the experiments led to useful failures, where 9 led to fail-stop failures and the remainder to non-fail-stop failures (although some also ultimately led to a fail-stop failure). Note that, we are only interested in predicting the first observable failures as afterward the system will already be impaired and at risk.

On the first take on the data, no augmentation was made. However, when reaching *Stage 4* of the process (i.e., *Tune Models per Failure/Class*), it was not possible to create accurate predictive models for the individual failures. After unsuccessfully expanding the set of ML algorithms and parameters, it was necessary to return to *Stage 2* (i.e., *Process, Cleanse, Augment Data*, this stage) to improve the information in the dataset. In the work discussed in *Section 6.2*, this data improvement was not necessary because the validation approach used was too optimistic (and ultimately not representative). However, using ELOOCV it quickly became evident that the raw dataset was not enough.

Although various techniques can be used to augment the data, this approach focused on feature extraction [Herff and Krusienski, 2019]. To consider the state of the system at a given time (on a per-second basis) different window-based statistical techniques were used. More precisely, the differential, average, and standard deviation for 2, 3, and 5-second windows, with and without the initial raw data, were explored. While these techniques enrich the datasets they also increase their complexity (i.e., the number of features is multiplied by each computed value) and thus it becomes necessary to establish a trade-off. For these experiments, the configuration that provided the best results was using the differential from the previous measurement combined with the current values. In practice, this considers the value of each feature as well as the details of how different it is from the previous observation and its signal (i.e., increasing if it is positive, decreasing if it is negative, or constant if its value is 0).

6.3.3 Parse Failures and Define Failure Classes

To create predictive models it is necessary to define the classes of the problem, that is, which classes of failures the models must predict (*Stage 3* of the proposed methodology). Ultimately, the goal is to predict whether or not a failure will occur, ideally distinguishing between different failure modes, at the very least between fail-stop and non-fail-stop failures (as this likely affects the severity of the preemptive measures).

As discussed in *Section 6.1*, the dataset contains two fail-stop types of failures, system *crash* and *hang*, which were considered as two different classes. For the remaining failures, due to the variety of non-fail-stop failures and their possible causes, this was not so trivial. This stage comprises the process of identifying, studying, and classifying the non-fail-stop failures (from the valid experiments) in classes. After thoroughly analyzing the data, 3 different high-level failure classes were identified: *i) memory-related* due to memory errors (e.g., *segmentation fault*), *ii) cpu/execution-related* due to errors at the cpu (e.g., *invalid opcode*), and *iii) kernel-related:* due to errors in the kernel (e.g., *recursive fault*). This taxonomy still requires further validation, but it will assist in the next stage for defining/validating which combinations of non-fail-stop failures should be assessed and tested first. Additionally, it will also be used to validate combinations that do not match this taxonomy. While it is relevant to create groups of failures that can be predicted, such groups must also be logical from a theoretical perspective (and thus one should not skip directly to *Stage 4*). The initial distribution of the experiments per failure class is: 6 *Crash*, 3 *Hang*, 10 *Memory*, 11 *CPU/Execution*, and 11 *Kernel*.

6.3.4 Tune Models per Failure/Class

The methodology proposed in this work addresses the task of identifying failure classes (to be predicted) as a mixture between top-down (e.g., creating an initial theory-based taxonomy) and bottom-up (predicting the different types of failure individually and then grouping them, identifying difficult/unpredictable failures in the process). This approach combined with ELOOCV allows identifying/validating the failure classes of the problem and estimating the performance of their respective models.

To provide some context, the predictive performance of the best model obtained using the traditional cross-validation approach (as discussed in *Section 6.2*) can be seen in *Figure 6.14*. As can be observed, it suggests that failure samples can be accurately distinguished from non-failure and that it is also possible to accurately distinguish between the different failure modes. However, when the same model was used to predict new unseen experiments (as occurs in a runtime environment), almost none of the failures were correctly predicted. In fact, that model was not able to generalize at all, as concluded when calculating the performance of the same model using ELOOCV (as can be seen in *Figure 6.15*, almost every sample was predicted as non-failure).

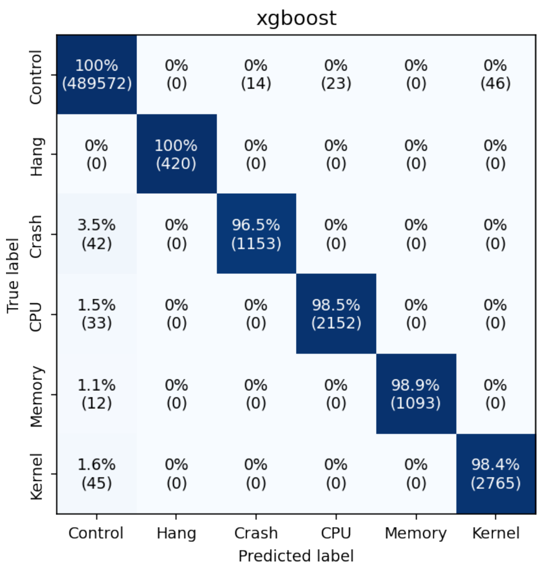


Figure 6.14: XGB., CV [20, 40]

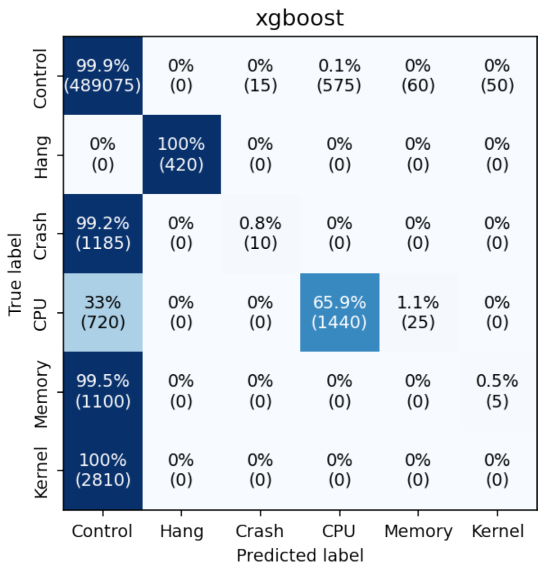


Figure 6.15: XGB., ELOOCV [20, 40]

In an attempt to create better models, the set of techniques was considerably expanded (e.g., Adaptive Synthetic (ADASYN) and Instance Hardness Threshold oversampling, Near Miss undersampling, LightGBM) even exploring deep learning approaches specific for tabular data/problems (e.g., TabNet [Arık and Pfister, 2020]). While this led to small improvements, it was still not possible to accurately distinguish between non-failure and failure samples. Thus, as proposed in *Stage 4* of the methodology, to simplify the problem, we decided to break it down by focusing on a single class at a time (i.e., developing separate predictors for each failure class). This quickly confirmed a hypothesis previously considered: different failures/classes display symptoms at different times and thus the best model for each one likely requires different parameters (i.e., $\Delta t_l, \Delta t_p$). This alone allowed creating models that could predict approximately 75% of failure samples and around 90% of non-failure samples. Still, these results were far from optimal, especially because when analyzing their performance experiment-wise, in *Stage 5*, there were several false-positives on golden runs (e.g., the best model for the *cpu* failure class raised 5 false-alerts on golden runs). This low performance, mainly on the non-fail-stop classes, could be due to two (main) reasons: there were unpredictable failures in the data (which could be adding noise and lowering the overall performance) or some failures were incorrectly grouped. Although initially a top-down approach was used (as referred to in *Stage 4*, e.g., by analyzing wrong predictions and trying to exclude/regroup) this was not time-effective. Thus, we found it necessary to use the alternative bottom-up approach.

Based on the results obtained in *Section 6.2*, and to limit the search space, we focused only on the algorithms that had shown promising results (i.e., tree-based: Bagging, RF, XGBoost). However, the search space of $\Delta t_l, \Delta t_p$, hyperparameters, and sampling techniques increased considerably. A brief list of the most relevant techniques and parameters can be seen in *Table 6.12* and *Table 6.13*, respectively. Several variations were considered for the sampling techniques such as different sampling ratios (i.e., 1:1, 1:2, 1:5), combined oversampling with undersampling, and in the case of the Near Miss algorithm, all its three versions (i.e., v1, v2, and v3) were used.

Table 6.12: Techniques Used in the Experiments

Process / Step	**Techniques**
$\Delta t_l / \Delta t_p$	[10, 20, 30, 40, 50, 60] / [20, 40, 60, 80, 100]
Sampling	Random under/oversampling, Near Miss, Instance Hardness Threshold, ADASYN, SMOTE
Algorithms	Bagging, RF, XGBoost

As proposed in *Chapter 4*, the first task was creating predictors for each type of failure (e.g., *segmentation fault*; experiments with identical failures must be predicted together, by a single model). While this led to accurate models, there was a considerable performance variation between multiple executions (e.g., the balanced accuracy of the best model for *segmentation fault* failures varied between 82% and 98%). By analyzing the models it was possible to observe that this was due to the undersampling process (all best models relied on undersampling techniques). As the undersampling was being done on all types of experiments (which included golden, non-failing, and other failing runs) this resulted in a high variation between multiple executions. As there are few failure samples, randomly

Table 6.13: Algorithms' Hyperparameters

Tech.	**Parameters**
RF	**estimators**: [100, 200, 500], **max_feat.**: [.1, .55, auto], **min_samp_leaf**: [.001, .01, 1], **min_samp_split**: [.001, .01, 2]
Bag.	**max_features**: [.1, .55, 1.0], **max_features**: [.5, .7, 1.0], **estimators**: [100, 200, 500]
XGB.	**booster**: ['dart', 'gbtree'], **rate_drop**: [0, .01, .1] **estimators**: [100, 200, 500, 1000, 1500] **gamma**: [0, 0.1, 0.3, 2, 5], **subsample**: [0.5, 0.7, 1] **min_child_weight**: [1, 3, 5], **colsample_bytree .**: [.5, .7, 1] **reg_alpha**: [0, 1, 3], **reg_lambda**: [1, 2, 5] **learning rate**: [0.5, .3, .1, .01], **max_depth**: [None, 3, 7, 15]

selecting the same number of samples from all the golden, non-failing, and other failing runs (i.e., to balance the training data set) meant that one execution could randomly select samples just from golden runs while the next execution could select all from other failing runs.

To minimize this variance, heuristic/algorithm-based sampling methods were used (e.g., Near Miss, Instance Hardness Threshold), but overall the results were also not satisfactory (e.g., for the *segmentation fault* failure, although more consistent, the balanced accuracy dropped to nearly 60%). Thus, we decided to train using only samples from golden runs and from experiments where the target failure mode was observed. This led to a noticeable increase in performance, as the models could now properly identify and predict the respective failure (e.g., once again, the balanced accuracy of the best model for the *segmentation fault* failure was now constant across the various executions, varying just between 97% and 98%). The only exceptions were the hang failures, which benefited from being trained with samples from all types of experiments (i.e., golden, non-failure, and other failure runs). While it is not possible to unequivocally say why, having more diverse samples in the dataset allows the algorithms to better model/distinguish hang failures. It should be noted that the performance of the models was assessed on every experiment (i.e., test set), including other failure runs.

Based on the initial high-level categorization and common characteristics between the best models for the different failures (e.g., the lead-time and prediction-window), we analyzed the performance using combinations of 2 types of failures. This then became an iterative process where the performance of these groups was analyzed and further combinations were made between the most promising and similar groups. During these iterations, the combinations were continuously compared and analyzed with regard to the initial taxonomy/categorization. This process was repeated until all the experiments belonged to a group/class. Any experiment not fitting in any of the existing groups/classes was analyzed to assess whether it was not possible to predict the failure observed or a new failure class was required. In the end, it was possible to create accurate models for the various failure classes, as can be seen in *Figure 6.16* and *Figure 6.17* (the models with the best and worst performance, respectively). Using this process led us to conclude that some of the failures were indeed miscategorized in the initial taxonomy (e.g., one of the failures at the *ext4_evict_inode* kernel function was being considered as kernel-related but it was in fact related to memory management) and were compromising the resulting models, as can be seen in *Table 6.14*. This was typically due to those experiments (mostly from the *kernel* class) having multiple failures, which were assessed as being due to one high-level cause but were indeed due to another. Nevertheless, most of the initial labels attributed to the experiments were kept. Additionally, this also allowed identifying three experiments (all due to the same fault) that could not be predicted. These failures occurred precisely when the workload was ending and it was likely due to some corruption that did not exhibit any symptoms.

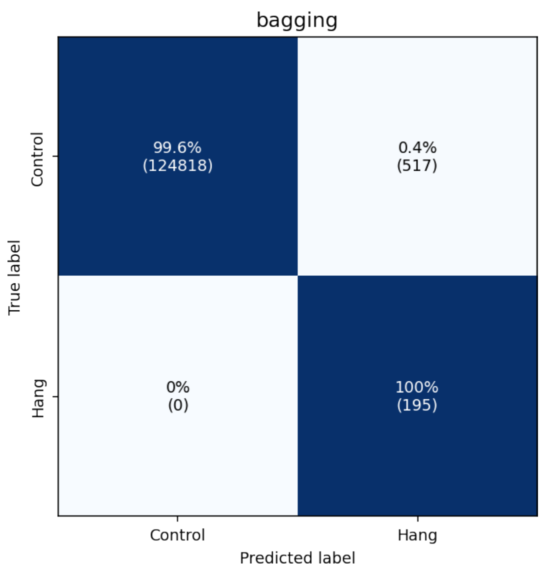


Figure 6.16: Hang, [30, 80]

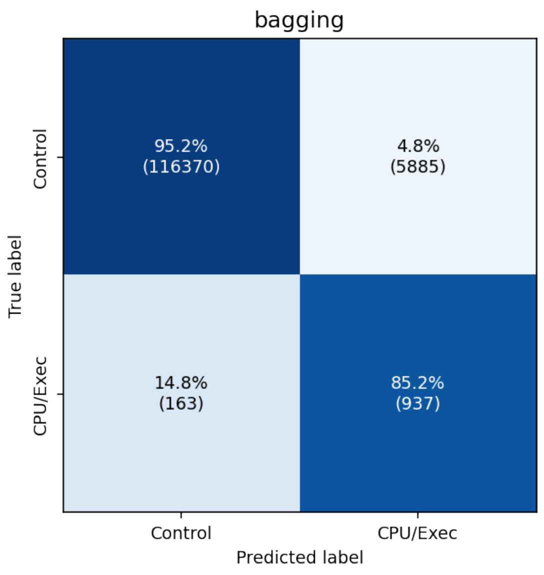


Figure 6.17: CPU/Exec, [60, 20]

Table 6.14: Experiments Distribution per Failure Class

Fail. Class		Initial	Refined	Excluded
fail-stop	Crash	6	6	0
fail-stop	Hang	3	3	0
non-fail-stop	Mem.	10	16	0
non-fail-stop	CPU	11	11	0
non-fail-stop	Kernel	13	4	3

6.3.5 Analyze Experiment-wise Performance

Although having models with high sample-wise performance is relevant, the end goal is to have models that can accurately and consistently predict incoming failures without (many) false-alerts (especially on non-failing runs). In practice, a system will never trigger preemptive measures based on a single alert, as that would be impractical and the chance of being a single false-positive is potentially high. As explained in *Chapter 4*, the logical approach is to only consider a prediction if there is at least a specific incidence (i.e., a specified number of alerts in a given time window, e.g., 4 alerts in the previous 5 seconds, as shown in *Figure 4.4*).

The top 3 models for each failure class (which ultimately were all obtained using the Bagging algorithm but with different parameters and $\Delta t_l, \Delta t_p$) were analyzed (as the model with the highest sample-wise performance is not always the best experiment-wise) to assess how many failures would not be detected and how many false-alerts would be raised. For demonstration purposes, a conservative requirement of having 5 consecutive alerts (i.e., 5 alerts in the last 5 seconds) was defined. The results can be seen in *Table 6.15.* As can be observed (in column *Missed Failure*), all the failures except one could be predicted. Additionally, with the exception of *crash* and *memory* failures, there were rather few misclassifications (i.e., failures being predicted as a different class) as can be seen in columns *Misclas. (non-fail-stop)* and *Misclas. (fail-stop)*). For the *memory* predictor, 6 of the misclassifications were in fail-stop failures and the remaining 7 in non-fail-stop failures (i.e., the predictor identified a failure run from other failure mode as being a *memory* failure). As no fail-stop failures were lost, these misclassifications are not problematic as those would also have been predicted by their respective (fail-stop) predictors. Regarding the 12 misclassifications in the *Crash* predictor, as they pertain to non-fail-stop failures this means that the system would have taken more conservative preemptive measures than strictly necessary. A key observation is that none of the predictors raised false-alerts in golden runs (35 golden runs were considered in these experiments), which is one of the most critical factors (i.e., avoid interrupting the system without need). Furthermore, as there were so few false-positives, in a practical scenario it would also be possible to define a smaller incidence (e.g., 3 alerts) to trigger the preparation of repair mechanisms to expedite their deployment if necessary.

It is worth noting that, to be consistent with the number of failures previously reported, these values pertain to a single execution of ELOOCV. Notwithstanding, considering the results for 5 executions with different random seeds (which means that the randomness of the process is different for each seed, affecting the sampling, algorithms, etc.) the conclusions are identical. This means that, out of 190 tests on failure experiments, **185 were predicted** and there was **not a single false-alert** on the 175 tests on golden runs, as summarized in *Table 6.16.*

Table 6.15: Experiment-wise Prediction Results

	Failure Class	Predict. Failure	Missed Failure	False Alerts (golden)	Misclas. (non-fail-stop)	Misclas. (fail-stop)
fail-stop	Crash	6	0	0	12	0
fail-stop	Hang	3	0	0	2	0
non-fail-stop	Mem.	15	0	0	**7**	**6**
non-fail-stop	CPU	10	**1**	0	2	0
non-fail-stop	Kernel	4	0	0	0	4

Table 6.16: 5-seed Experiment-wise Summary

	Experiments	**Missed (False-Negatives)**
Failures	190	5

	Experiments	**False-Alerts**
Golden	175	0

6.3.6 Deploy Best Model per Failure Class

The last stage of the methodology comprises the deployment and continuous monitoring of the predictors on the target system taking into consideration the details presented in *Chapter 4*. This allows assessing how the performance of the failure predictors holds on the operational scenario and whether corrective measures are necessary. As there is no real operational scenario for the models developed in this evaluation, this stage could be studied by conducting lengthier campaigns, both with and without fault injection, ideally also exploring/injecting different/new faults. Notwithstanding, this falls out of the scope of this study and is considered for future work.

6.3.7 Discussion

As discussed before, not taking into consideration the various stages and issues discussed in our methodology for developing predictive models (e.g., identifying false-predictors in the data due to the synthetic nature of the experiment, or using inadequate techniques to assess the performance) leads to predictors that appear to have modeled the problem but will ultimately fail if deployed on real systems. As an example, the models developed in *Section 6.2* were developed taking into consideration the recommended ML techniques but ultimately could not predict any new unseen failures. Although the reason why this occurs was not formally validated, the hypothesis is that modern ML algorithms are so effective that they are able to identify patterns in each failing sample that allow them to predict other samples from the same run when shown in the test set (as discussed in *Section 2.2*, one of the risks/limitations of traditional cross-validation is that it is possible to have samples from 'past' and 'future' in both the train and test sets). According to the literature, this should not be such a critical issue, but given the experiment-based nature of OFP (and its failures), we have shown that it is not a suitable approach for OFP. One of the reasons why this was not detected earlier is that many of the studied algorithms did not perform well, and it was only after a long experimental campaign and fine-tuning that it was possible to have accurate models (i.e., these 'patterns' or 'leaks' were not evident). ELOOCV tries to address this issue by testing on entire experiments instead of individual/isolated samples. This quickly allowed observing that each failure class required specific ML methods and parameterizations (e.g., *hang* failures could be better predicted with a lead-time of 30 seconds while *cpu/exec* with 60 seconds). In the end, although it was often necessary to return to previous stages to either expand the set of techniques or enrich the dataset, it was possible to create accurate predictors.

Another complex issue that the methodology also tries to address is defining the classes of the problem (to be predicted). While fail-stop failures are easy to categorize (typically there will be a distinct class for each), the identification and categorization of non-fail-stop failure modes are dependent on the target system, and as a result, cannot be completely generalizable. In such situations, the typical approach is to analyze the different failures, identify their root/likely cause, and group them accordingly (only if logical from the perspective of the system). Still, there is no silver bullet for this, and thus the resulting classes may not be ideal for prediction (e.g., some failures may be mistakenly grouped together, others may have no noticeable/relevant symptoms and therefore cannot be predicted). To detect these situations, the proposed methodology uses a combination of a *top-down* and *bottom-up* approach that allows identifying problematic failures and predictable classes following a predefined theory-based taxonomy. In this study, this allowed identifying some failures that were initially placed in an inadequate class as well as some failures that could not be predicted. As divergences should be analyzed, in this case, this was because such experiments had multiple failures and the cause had been wrongfully interpreted. Nonetheless, some symptoms of non-fail-stop failures may overlap with symptoms of fail-stop failures (e.g., a memory leak which may, or may not, lead to a system hang), leading to some false-positives on different failure modes. This is ultimately inevitable and should be analyzed as an overall process of predicting different failure modes (and severity) on complex systems and how it can be used to improve their reliability.

The proposed methodology also focuses on real-world usage scenarios and goes beyond sample-wise prediction (the typical approach in ML problems) by considering experiment-wise performance. While sample-wise performance is relevant to guide the search for better models, when selecting the predictors for deployment the goal is also to have models that are consistent at predicting the failures and not giving false-alerts (as no system will halt execution due to a single alert). In this study this led to the conclusion that the model with the highest sample-wise performance is not always the best when considering experiment-wise predictions (although it could have fewer false-positives, they may be spread across more experiments or with a higher incidence per experiment). In the end, it was possible to create accurate models for each failure class (i.e., it only failed to predict one *cpu/exec* failure, as shown in *Table 6.15*) without raising false-alerts on golden runs.

Given the complexity of the problem and the dependence on its nature/data, some stages of the proposed methodology are somewhat abstract, that is, it is not a strictly-defined process that the user can blindly follow. However, this was never the purpose of the methodology (otherwise it would have to be specific to a certain OS or failure modes) and therefore should not be seen as a limitation. The six stages provide instructions on how to generate failure data using fault injection and on using them to create accurate representative predictive models. It contains several feedback loops that instruct the user to the need to return to previous stages to improve the process if necessary.

Finally, due to the large number of ML algorithms, techniques, and hyperparameters, combined with a large number of experiments and failures, the methodology guides the process on an iterative basis. Ultimately, this means that it will not exhaust every possible combination. While other specific parameterizations could lead to slightly better results, the iterative process and feedback loops encourage the search for better and more accurate models. Notwithstanding, within the model selection process (i.e., fine-tuning the hyperparameters) the user may choose to use either an exhaustive grid-search approach or an algorithm/heuristic-based approach (e.g., Bayesian optimization). Additionally, the instantiation of the methodology focused on a single workload. While we currently do not have definite results, a preliminary analysis on applying the proposed methodology to other workloads led to observations that are consistent with the case study presented, confirming the viability and benefits of using the proposed methodology.

6.4 Benchmarking Failure Predictors

Besides developing accurate failure predictors, effectively implementing failure prediction requires an adequate selection of the most suitable models using appropriate metrics and properly comparing them using adequate procedures. This section demonstrates how the benchmarking approach proposed in *Chapter 5* can be used in practice. It is used to compare several failure prediction solutions, including multiple algorithms, preprocessing techniques, and configurations, for different problems and benchmarking tasks. Overall, it details the steps of the approach and how they should be followed to assure that the best model is chosen for the intended problem and scenario. It also explores and highlights the importance of assessing the robustness of the models against small variations in the data and overviews some techniques to overcome this.

The Propheticus tool (introduced in *Section 3.3*) was used to train and assess the performance of the ML algorithms (phases *3 - Execution of Prediction Algorithms* and *4 - Metrics Calculation, Assessment, and Comparison*, of the procedure), as it provides an environment that allows attaining some of the required properties (e.g., repeatability, portability). For compatibility, the remaining phases (i.e., *1 - Preparation*, *2 - Dataset Building, Validation*, part of *4 - Metric Calculation, Assessment, and Comparison*, and *5 - Robustness Assessment*) were implemented as extensions within Propheticus. Nonetheless, the proposed approach is conceptual, and therefore can be used with other implementations. This analysis focuses on the *cpu* workload, a cpu-intensive workload representative of common real-world usages (e.g., video-editing, computation).

6.4.1 Preparation

The definition of the algorithms, techniques, and respective parameters is one of the most relevant parts of step *1 - Preparation.* To provide a comprehensive study, several options were considered for this experiment, as can be seen in *Table 6.17.* Based on the results discussed in the previous sections, this analysis focuses on tree-based algorithms as they performed consistently better than the alternatives (e.g., SVM and NN were not able to achieve similar performances). A grid-search approach was used to fine-tune the models considering the hyperparameters listed in *Table 6.18.* In the end, the task at hand is to identify and rank the best solutions for our problem (for each failure mode) depending on the business needs (the scenarios) and assess and improve their robustness to small variations in the data.

The results obtained in *Section 6.2* and *Section 6.3* have shown that in some configurations sampling techniques could lead to better models. To explore this, a more sophisticated oversampling method, *ADASYN*, was included, and also several different sampling ratios were considered (as shown in the literature, different sampling ratios have a considerable impact on the resulting models [Kim and Kim, 2018]). Additionally, the influence of combining different sampling techniques (i.e., oversampling followed by undersampling) was also studied. A Z-score standardization was applied to the features. As the dataset contains a considerable number of features, two common feature selection methods were used: *variance*, removing features with 0 variance, and *correlation*, removing features with correlation above 90%.

Another pertinent decision included in the preparation step is the definition of the pair $\Delta t_l, \Delta t_p$. The results presented in the previous sections suggest that different failure modes likely have different ideal $\Delta t_l, \Delta t_p$, the premise being that different failures start to manifest at different times. Thus, for this study it was decided to focus on predicting each failure mode individually and exploring the combinations of the $\Delta t_l, \Delta t_p$ values shown in *Table 6.8.* Ultimately this corresponds to different benchmarking experiments to identify the best solutions for each failure mode (~42336 different combinations per mode).

The final task is the definition of the usage scenario (as detailed in *Section 5.1*). This should portray the behavior of the environment where the system will run, to guide the search for the best model. To demonstrate how the best model varies according to the requirements of the system, all the scenarios previously described are analysed.

Table 6.17: Techniques Used in the Experiments

Process / Step	**Techniques**
$\Delta t_l, \Delta t_p$	[20, 40, 60]/[20, 40, 80]
Scenarios	*High/Medium/Minimum-criticality*
Feature Selection	Variance, Correlation
Sampling	Random under/oversampling, ADASYN
Sampling Ratios	Over.: [2, 4], Under.: [1, 10], Over. and Under.: [(2, 1), (2, 5), (4, 1), (4, 5)]
Algorithms	DT, Bagging, RF, XGBoost

Table 6.18: Algorithms' Hyperparameters

Alg.	**Hyperparameters**
DT	**min_samp_split**: [001, .01, 2], **max_feat.**: [.1, .55, None], **min_samp_leaf**: [.001, .01, 1]
RF	**estimators**: 100, **max_feat.**: [.1, .55, auto], **min_samp_leaf**: [.001, .01, 1], **min_samp_split**: [.001, .01, 2]
Bag.	**max_features**: [.1, .55, 1.0], **estimators**: 100 **max_samples**: [.1, .55, 1.0],
XGBs.	**estimators**: 100, **learning_rate**: [.1, .3], **max_depth**: [7], **subsample**: [0.7, 1], **min_child_weight**: [1, 5], **colsample_bytree**: [.7, 1]

6.4.2 Dataset Building and Validation

Dataset Building and Validation is intended to generate the dataset/workload to benchmark the different solutions. In practice, as this work focuses on predicting a single failure mode at a time, what happens is that only the experiments of the target failure mode are considered as 'positive/failure'. All other experiments, even other failure runs, are considered as 'negative/non-failure'. Additionally, due to the number of combinations for the ML techniques and parameters, as well as the number of iterations required for ELOOCV (which conducts a train/test iteration for each experiment), experiments where faults were injected but no failure was observed were not included in the dataset (i.e., we focus on the golden and failure runs).

The datasets were labeled according to the pairs $\Delta t_l, \Delta t_p$ chosen in step *1 - Preparation* (as proposed by Salfner et al. [Salfner et al., 2010]). As an example, for the pair [20, 40], the resulting datasets have approximately 18000 non-failure samples and 69, 239, 437, 435, and 153 *hang, crash, cpu/exec, memory,* and *ker-*

nel samples, respectively. Briefly revisiting, a sample contains the values of the features (i.e., system metrics) for a given instant t_{sample}. Samples for which no (target) failures were observed within the interval $[t_{sample} + \Delta t_l, t_{sample} + \Delta t_l + \Delta t_p]$ are known as *non-failure* samples, and *failure* samples otherwise. As it is possible to observe, the number of failure samples is considerably low, hence the need to carefully select which techniques and metrics to use. It is also possible to see that there are fewer samples of fail-stop failures (i.e., hang, crash) compared to the non-fail-stop cases. This is mostly due to the fact that only the first detected failure is considered, as most fail-stop failures were typically preceded by non-fail-stop failures.

Another task of this step consists of validating the datasets. By analyzing them, it was possible to observe that there were enough failure samples for each failure mode (i.e., there must be at least 2 experiments of each failure mode, one for training and the other for testing), there were no missing data, the number of features was the same for all samples, and each feature contained only one data type.

6.4.3 Execution of Prediction Algorithms and Metrics Calculation, Assessment, and Comparison

Moving to step *3 - Execution of Prediction Algorithms*, a grid search was conducted for each algorithm and respective hyperparameters using ELOOCV. Each configuration was run 30 times with different random seeds, leading to the metrics calculation in step *4 - Metrics Calculation, Assessment, and Comparison*. The comparison and ranking of the resulting models were done considering the best model found for each algorithm. The top three solutions (i.e., algorithm, techniques, and respective sampling ratios) for each of the failure modes and criticality scenarios can be seen in *Table 6.19*. For each solution, the value of the performance metric is presented: *informedness*recall*, *f-measure*, and *markedness* for the *high-*, *medium-* and *minimum-criticality* scenario, respectively.

The best solution for each of the failure modes and criticality scenarios is obtained using different methods. Although XGBoost is a state-of-the-art algorithm, its performance was identical or lower than the other algorithms for various failures modes. Still, XGBoost has several parameters and it is possible that the chosen set was not enough to completely leverage its potential.

In general, the best solutions for the *high-criticality* scenario were obtained by using oversampling techniques with different ratios, often combined with undersampling (e.g., for the *memory* failure mode). Also, while DT achieved remarkable results (especially considering its simplicity), it did not match its ensemble versions. Another observation is that the best models are often achieved using different sampling ratios (e.g., 1st and 2nd for the *kernel* failure mode and *high-criticality* scenario). Different ratios ultimately influence the resulting model (and its performance), as has already been studied in other domains (e.g., [Rácz et al., 2021]). Many of the best models for some failure modes relied on feature selection by correlation (e.g., *crash*, *hang*) while others not so much (e.g.,

Table 6.19: Rank per Failure Mode/Scenario

Scen.	Rank	1st		2nd		3rd
crash	*High*	RF + ROS (2) (0.965)	>	XGB. + ROS (2) + Corr. (0.954)	>	DT + Corr. (0.943)
	Med.	RF + Corr. (0.563)	>	Bag. + ROS (2) (0.434)	>	DT + ROS (2) + RUS (1) + Corr. (0.275)
	Min.	Bag. + Corr. (0.991)	>	RF + ROS (2) (0.457)	>	DT (0.195)
hang	*High*	Bag. + RUS (1) + Corr. (0.999)	>	RF + Corr. (0.998)	>	XGB. + ADA. (4) + Corr. (0.994)
	Med.	XGB. (0.993)	>	Bag. + ROS (4) (0.989)	=	RF. + ROS (4) + Corr. (0.981)
	Min.	XGB. (0.999)	>	Bag. + ROS (4) (0.999)	=	RF + ROS (4) + Corr. (0.971)
cpu	*High*	Bag. + ADA. (2) + RUS (1) + Corr. (0.918)	>	XGB. + RUS (1) + Corr. (0.883)	=	RF + RUS (1) + Corr. (0.864)
	Med.	Bag. (0.758)	>	RF (0.638)	>	XGB. + ADA. (4) (0.552)
	Min.	RF (0.991)	>	Bag. + Corr. (0.962)	>	DT (0.722)
memory	*High*	Bag. + ADA. (4) + RUS (1) (0.960)	=	RF + ADA. (4) + RUS (5) (0.956)	>	XGB. + ROS (2) + RUS (1) (0.949)
	Med.	Bag. (0.742)	>	RF + Corr. (0.714)	>	DT (0.675)
	Min.	Bag. + ADA. (4) (0.995)	>	RF + ROS (2) (0.810)	>	DT (0.651)
kernel	*High*	RF + ADA. (2) + Corr. (0.982)	>	DT + ADA. (4) (0.970)	>	Bag. + ROS (4) + RUS (5) + Corr. (0.964)
	Med.	RF + ADA. (2) (0.331)	>	Bag. + ROS (4) + Corr. (0.187)	>	XGB. (0.181)
	Min.	RF + ADA. (2) (0.266)	>	Bag. + ROS (4) (0.103)	>	XGB. (0.100)

Legend:
$a > b$: a has significant differences from b
$a = b$: a does not have significant differences from b
(#.###) : value of the performance metric according to the scenario
RF=Random Forest (RF); **Bag.**=Bagging; **XGB.**=XGBoost; **DT**=Decision Tree (DT); **ADA.**=ADASYN; **ROS**=Random Oversampling; **RUS**=Random Undersampling; **Corr.**=Correlation (feature selection)

memory, *kernel*). This suggests that some features may have relevant information for specific failure modes. It is also possible to observe that, for the *medium* and *minimum-criticality* scenarios, almost none of the best solutions included the use of undersampling techniques. This is due to the metrics considered in these scenarios, which are more concerned with false-positives (which typically increase with the use of undersampling).

Analyzing the performance of the models for the different scenarios allows understanding how the different metrics determine which one is the best. As an example, for the *memory* failure mode and the *high-criticality* scenario, the best model (obtained using Bagging with ADASYN oversampling and Random Undersampling, with a computed metric of 0.960) correctly predicted 100% of the failure samples and 92% of all non-failure samples (due to the imbalance in the data, 8% of false-positives is already considerable). On the other hand, for the *medium-criticality* scenario, the best model (created using only Bagging, with an f-measure of 0.742) correctly predicted 99.8% of non-failure samples, but only 66.3% of failure samples (a considerably lower number of false-positives, at the expense of missing some failures). A similar observation can be made for the best model for the *high-criticality* scenario for *crash* failures (obtained with RF and Random Oversampling, with a computed metric of 0.965), which was able to correctly classify 94.6% of non-failure samples and 99.5% of failures. For the *medium-criticality* scenario, however, the best model (generated using RF and feature selection by correlation, with an f-measure of 0.563) correctly classified 97.6% of non-failure samples, but only 88.4% of failure samples. It is also possible to observe that both *crash* and *kernel* failures have solutions with low performance on the *medium-* and *minimum-criticality* scenarios. A detailed analysis of the models allowed concluding that they raise many false-positives (especially compared with the small number of failure samples) and thus the low performance on those scenarios.

Another interesting observation is that the $\Delta t_l, \Delta t_p$ of the best models varied for each failure mode, algorithm, and criticality. As an example, while for the *hang* failure mode the best Bagging model was obtained using the pair [40, 40], for *memory* failure mode it was obtained using [60, 40]. It also appears that *hang* failures are easier to predict than the remaining failure modes. The best model for the *high-criticality* scenario (created using Bagging and Random Undersampling with feature selection by correlation, with a computed metric of 0.999) could predict 99.7% and 100% of non-failure and failure samples, respectively. Nonetheless, the current dataset contains few hang failures and these results need further validation.

Although the training time was not taken into consideration in the ranking (as this study used multiple machines and the implementation of the benchmark is not concerned with optimization but rather flexibility), it was possible to observe that, for the same sampling and preprocessing techniques, XGBoost took longer than the other algorithms (e.g., for the *cpu* failure mode, without sampling and just removing constant features, each fold of XGBoost took around 50 seconds while Bagging took 5 seconds). Notwithstanding, this is highly dependent on several factors, such as the computational power available (the machine used for these comparisons has 48 cores and thus algorithms such as Bagging and RF can leverage parallelization), the implementation of the algorithms, and the (hyper)parameters.

To state that a given solution is better than others, statistical comparisons are required. The results for the top-ranking models can also be seen in *Table 6.19* (i.e., $a > b :$ a has significant differences from b; $a = b :$ a does not have significant differences from b). For the cases where there were no significant differences between the best models (e.g., for the *memory* failure mode and *high-criticality* scenario) the tiebreaker metric was considered, although in this case it did not cause any changes. To illustrate, let's take as an example the top three models for the *high-criticality* scenario for the *crash* failure mode. Because the dataset is the same for all experiments, a paired statistical test was used. To decide between parametric and non-parametric tests, the normality of the data (using the *Lilliefors* and *Shapiro-Wilk* test) and the homogeneity of variance (using the *Levene* test) were analyzed. None of the conditions were satisfied, so the *Friedman's ANOVA* was used [Field, 2013], which gave a $p\_value = 0.0005$. It is safe to state that there are significant differences between at least two models for a significance level of 5% ($p\_value < 0.05$). To identify the differences, a *post-hoc* analysis was done using the *Bonferroni* correction for multiple comparisons. This identified significant differences between all models, i.e., between the 1st (RF) and the 2nd (XGBoost), as well as the 2nd and the 3rd (Bagging).

6.4.4 Assessing Robustness

The last step of the benchmarking framework, *6 - Robustness Assessment*, is the assessment of the performance of the models against small variations in the data. Depending on the number of samples and chosen techniques, assessing and enhancing the robustness of the models can take some time, thus, this analysis focuses on the XGBoost algorithm and the *cpu* failure mode (because it has the highest number of samples). Although XGBoost did not lead to the best solutions (probably because it required more tuning), it achieved good results overall, often matching the performance of the best models. Additionally, it is a state-of-the-art algorithm and thus it is relevant to assess how sensitive its resulting models are. Moreover, a novel robust algorithm based on XGBoost has been recently proposed [Chen et al., 2019a], which allows for a better comparison.

There are several white-box methods to generate adversarial samples for differentiable algorithms (such as NN) but for non-differentiable algorithms (such as XGBoost) there are not many alternatives. Thus, we decided to use a state-of-the-art black-box algorithm, the *HopSkipJump*-attack [Chen et al., 2020] (an advanced version of the boundary attack [Brendel et al., 2018]), which relies only on the final class prediction. While there were other approaches available (e.g., GANs [Goodfellow et al., 2014]) such solutions are not straightforward to implement and would introduce additional challenges (e.g., how to validate that the generated samples are representative or realistic of a system state of a given class for OFP). The L_∞-norm (which measures the largest variation) was used to measure the perturbations considering a maximum perturbation of 0.3. To understand the magnitude of such a change on the non-standardized values, the maximum perturbation was calculated on different features. As an example, for the *average cpu percentage of system apps* feature, whose values range from 0% to 88%, it would mean a maximum change of 0.81%. Regarding the *average apps memory use (MiB)*, which ranges from 32MB to 505MB, it would mean a maximum variation of 5.12MB. Thus, while dependent on how the values of the features are distributed, with Z-score standardization a perturbation of 0.3 is a conservative value regarding expected variations in the data.

As can be seen in *Table 6.20*, it was possible to generate 7277 adversarial samples out of the 24671 samples considered, with an average distortion of 0.185 within a maximum perturbation of 0.3. The balanced accuracy (this metric is used for this analysis as it is easy to understand what it means in terms of the predictive performance, while also considering the imbalance in the data) dropped from 92% to 57% (i.e., when replacing the original with the adversarial samples) and the recall (i.e., number of correctly classified failures) dropped from 94% to 53%. To assess the minimum/maximum variations, the normalized values were reverted to their original scales. For the sample with the minimum perturbation, the largest variation was merely changing the feature *average of microseconds lost in iddledjitter* by 6.61e^{-5} (its values range between 494 and 31986). For the sample with the maximum perturbation, the largest variation was for the feature *average of page faults per second* by a value of 220 (its values range between 0 and 17897).

Table 6.20: Linux Unrobust Model vs Adversarial Samples (max $L_\infty = 0.3$)

# Adv. Samples	**7277** out of 24671
Avg. Perturbation	0.185
Adv. Balanced Accuracy	From 92% to **57%**
Min. Perturbation	*average of microseconds [...]* **6.61**e$^{\mathbf{-5}}$ - [494, 31986]
Max. Perturbation	*average page faults [...]* **220** - [0, 17897]

The previous approach provides a good estimate of the performance of the model in the presence of adversarial samples, but it does not give any formal guarantees, which is often necessary for safety critical systems. To achieve this, formal verification methods have been developed to calculate a lower bound of adversarial perturbations. Once again, while there have been various verification methods developed for NN, tree-based models (including ensembles) have not been as widely researched. Still, Chen et al. [Chen et al., 2019b] proposed a tree-based robustness verification method that scales well to large ensembles. For the previous model and samples (i.e., XGBoost for the *cpu* workload), the method identified a minimum adversarial distortion lower bound average of 0.298, with a mean verified error of 54% (i.e., a guaranteed upper bound of error for a maximum L_∞ perturbation of 0.3).

While this model is still susceptible to variations in the data, its robustness is considerably higher than the models obtained in *Section 6.2* (developed/selected using k-fold cross-validation). As an example, for those models the adversarial samples had an average distortion of 0.066 and the performance dropped from 99% to 19% (and the recall to 7%). The tree verification method identified a minimum adversarial distortion lower bound average of 0.094, with a mean verified error of 100%. Although it was not possible to determine exactly why the models were so susceptible to changes in the data, it appears to be related to the fact that they were predicting multiple failure modes (i.e., multi-class failure predictors). The decision boundaries between different failure modes are likely more complex and closer to the various samples, therefore only requiring minor perturbations to be misclassified. It is also possible that ELOOCV leads to less overfit models and thus samples are not as close to the decision boundaries. Notwithstanding, even with standard cross-validation single-class failure predictors were considerably more robust than their multi-class counterparts.

By analyzing the robustness of the models for the remaining failure modes it was possible to observe that some are quite robust by default. As an example, for the best XGBoost model for the *memory* failure mode and the *high-criticality* scenario, it was only possible to generate 713 adversarial samples (under the max perturbation of 0.3) out of 24930. Furthermore, the verification method identified a minimum adversarial distortion lower bound average of 0.902, with a mean verified error of 7%, both of which indicate a very robust model.

As previously shown, the best XGBoost model for the *cpu* failure mode in the *high-criticality* scenario is sensitive to small variations in the data. However, there is no single technique or silver bullet for creating more robust models (as briefly discussed in *Section 2.2*). In this direction, we decided to test the approach proposed by Chen et al. [Chen et al., 2019a] (a novel robust algorithm based on XGBoost). Adversarial samples were generated to validate the robustness of the model, and the results were considerably better. For this case, it was only possible to generate 1646 adversarial samples (against the previous 7277) with an average perturbation of 0.198. The balanced accuracy on the final set was 88%, and the formal verification identified a considerably larger minimum distortion average of 0.422, and the guaranteed error rate we reduced to 22%. More precisely, the recall of the resulting model was kept at 94% (from 99% on the benign samples, the robust model had slightly better performance than the natural XGBoost, possibly due to a more extensive fine-tuning of the hyperparameters). It is important to note that even when using more robust algorithms, the resulting model may still be susceptible to perturbations in the data. This can likely be mitigated by training the model to be robust to higher perturbations (thus tolerating better lower variations). Nevertheless, as previously argued, more robust algorithms can come at the expense of predictive performance and thus a trade-off must be defined considering the needs of the system.

The number of samples per distortion/model can be seen in *Figure 6.18*. As was previously stated and can be observed, for the non-robust model it was possible to generate many adversarial samples (i.e., 7277 out of 24671) within the maximum perturbation. Contrarily, for the robust model, there were significantly fewer adversarial samples (i.e., 1646 out of 24671) and many were close to the maximum perturbation.

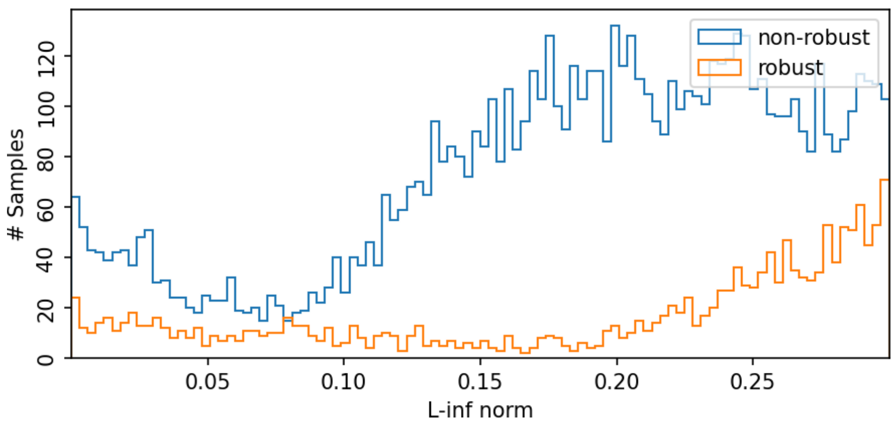


Figure 6.18: Number of Samples per L_∞ Perturbation (up to 0.3)

6.4.5 Discussion

The proposed framework allowed conducting a thorough and accurate campaign through well-defined steps and procedures that support properly benchmarking alternative models for different OFP tasks, demonstrating that different contexts (e.g., usage scenarios, failure modes) require different solutions. This corroborated the fact that each failure mode has different ideal $\Delta t_l, \Delta t_p$, and therefore having a single multi-class failure predictor is not likely/possible. It was also possible to observe that there are differences between the best models for each scenario. These results emphasize the need for a structured procedure to compare and rank solutions. They also highlight the need for a clear definition of the requirements of where the models are to be deployed (which in turn determines the metrics that should be used) as well as the need to create them using data that pertain to the system where they will operate. Following the proposed framework it was also possible to ascertain that the best models were susceptible to minor perturbations in the data. Different approaches (e.g., single-class *vs* multi-class predictors) may present different robustness to such changes, which should be taken into account when selecting the best solutions. This can be identified and mitigated using our benchmarking framework, leading to more robust models.

The goal of the framework and of the analysis presented is to demonstrate how the performance of the algorithms varies under different conditions, how complicated it is to choose the best model for different scenarios, how sensitive ML models are to imperceptible variations in the data, and how the proposed framework can assist in this task. The analysis does not dwell on the specific performance of the models as the focus of the proposed approach is on comparison and ranking among the different solutions. Additionally, although some of the concepts considered in the framework have already been used in different contexts, such as scenarios and metrics, they have not yet been studied and defined for the OFP problem. Also, while most of the metrics detailed in this work are well-known, what they actually represent and measure (and their drawbacks and limitations) are not known to many of the researchers that use them (that is why so many studies use inadequate metrics for assessment and comparison).

Benchmarking is usually seen as expensive and laborious. Concerning the time needed for the benchmarking process, it was very low, mainly because it can be mostly automated. In fact, after having the framework pre-requisites met (e.g., dataset, scenarios), the effort needed to benchmark additional systems is rather small. Regarding *ease of installation and use*, Propheticus can be instantiated and used to implement the benchmarking procedure (and facilitate the step of exporting and validating the data). Regarding the robustness verification step, this is partially dependent on the selected algorithms. Nonetheless, many of the current state-of-the-art solutions provide Python implementations, which can be easily integrated with Propheticus. Still, the user may use other implementations, although this will dictate the ease of installation and use. Concerning *promptness*, most experiments were completed in less than a couple of hours. This property is directly related to the dimension of the datasets, as many methods have an exponentially increasing time complexity with the size of the dataset, and thus it is not a constraint of the framework itself. Additionally, the framework is *not intrusive* as it does not require any kind of modification to the failure prediction models. Regarding *portability*, it can be run in any system and using data pertaining to any environment (as long as they are in the expected structure) and it can be used to assess and compare any kind of failure prediction model. Concerning *repeatability*, using ELOOCV assures that the results are not biased by the data split or data leaks, and the 30 executions with different seeds provide statistical support. The *representativeness* of the benchmark is related to the dataset, which is provided by the user. The datasets used were generated using a thorough fault injection campaign, which currently is the best accepted alternative to generate realistic failure data. Additionally, the robustness assessment provides further assurance regarding the performance of the algorithms when dealing with minor variations of the data.

6.5 Summary

This chapter demonstrated and overviewed in detail the most relevant steps required to develop predictive models for OFP on a new system. It highlighted how the different contributions proposed in this work can be used to achieve this, and their importance and relevance to assure reproducible and reliable results while providing confidence that results will hold in an operational scenario.

The experimental evaluation iterated through the various necessary steps and tasks, from generating the failure data in *Section 6.1*, to creating predictive models in *Section 6.2* and *Section 6.3*, to benchmarking alternative predictive solutions in *Section 6.4*. Within each task, a detailed analysis was presented on the influence and relevance of each contribution. *Section 6.1* thoroughly explored the guidelines on how to properly implement a testbed, as well as the use of fault injection to generate realistic failure data. *Section 6.2* used the generated data to explore the viability and influence of many different ML algorithms and techniques on creating accurate failure predictors. *Section 6.3* followed the methodology proposed in

Chapter 4 to thoroughly process the generated data to develop accurate failure predictors, leveraging the ELOOCV approach (also proposed in *Chapter 4*) to provide a more realistic assessment of the performance of the models. It also included a comparison with the solutions developed in *Section 6.2*, which were created using standard ML techniques. Finally, *Section 6.4* used the benchmarking approach introduced in *Chapter 5* to properly benchmark alternative predictive solutions for OFP. It explored the concept of scenarios and how they influence the most suitable models taking into consideration the technical needs of the target system, also addressing the need to explore the robustness of the models to variations in the data, which is directly related to the confidence in the results obtained using the benchmark. Various comparisons with related work were also provided throughout the section, on how the results differ or how the process leads to more realistic insights.

Although not detailed in this section, the ability to create failure predictors for the Windows OS was also analyzed. This provides further assurances and external validity concerning the generalizability of the results and processes, suggesting that it is likely possible to develop accurate failure predictors for similarly complex systems. The details of this case study can be found in *Appendix A*.

Chapter 7

Conclusions and Future Work

The pervasiveness of software systems in critical everyday tasks requires them to be highly reliable. However, software complexity has also grown considerably in recent years and it is now almost impossible to detect all faults before deployment. Several techniques have been developed to address this issue. OFP is one of such techniques, a fault-tolerance approach to predict incoming failures in the near future, using past data and the current state of the system. Notwithstanding, developing accurate failure predictors is a complex task, and as a result, OFP is still not widely used or researched nowadays.

The work presented in this thesis tries to overcome the most pressing issues and limitations to the widespread use of OFP. This thesis introduced a comprehensive framework to support OFP, composed of various interconnected elements, each addressing a different issue or limitation. These components are divided into two groups: *techniques and artifacts* and *procedures and methodologies*. The first group comprises three main contributions to support the research and development of OFP solutions: *i)* detailed guidelines and reflections on configuring and deploying a *testbed for dependability experiments*, *ii)* a well-defined process on how to use *fault injection to generate failure data*, and *iii)* a flexible ML toolbox that includes the functionalities required to *develop accurate predictive models* for OFP. The second group encompasses two major contributions to create accurate ML-based OFP solutions for a given target system: *i)* a detailed *multi-stage methodology to develop predictive models* that takes into account the specific characteristics of combining fault injection and ML to create predictive models, and *ii)* a *conceptual framework for benchmarking predictive solutions* for OFP, ensuring a sound assessment and comparison of failure predictors while considering the technical needs of the target system. These contributions advance the state of the art on OFP and reduce the existing gap in the literature, by providing detailed and well-defined artifacts, guidelines, methodologies, and processes to develop failure predictors, taking into consideration the distinctive characteristics of the problem and where the predictors will operate.

To demonstrate how the proposed contributions can be used in practice to develop accurate failure predictors for a complex system a thorough experimental evaluation was conducted. It comprised an extensive fault injection campaign targeting the Linux OS, considering multiple workloads representing different usage scenarios, several fault types, and various failure modes. The experimental evaluation overviewed the use and relevance of each contribution, from properly implementing a testbed, using fault injection to generate failure data, adequately developing and assessing the performance of failure predictors, to objectively comparing and ranking different solutions according to the needs of the system. It assessed and thoroughly analyzed the viability and performance of several ML methods, ranging from classic to state-of-the-art approaches, which demonstrated that it is possible to develop accurate failure predictors for modern systems.

Several insights can be taken from the results obtained throughout the experimental process, such as the importance of exploring different algorithms and techniques that take into account the characteristics of the data. Exploring different usage scenarios also highlighted the need to consider the technical requirements of the systems where the predictors will operate, which in turn dictate which models are best and which metrics should be used to assess their performance. This evaluation also highlighted the necessity of considering the specific characteristics of the problem when assessing the performance of the models to avoid unrealistic estimates. Overall, DT-based algorithms (e.g., RF, XGBoost) were able to achieve significantly better results than the alternatives. The need to assess, and improve, the sensitiveness of the models to variations in the data was also studied, a pressing current limitation of ML models. Ultimately, the experimental evaluation demonstrates that not only is it possible to create accurate failure predictors but that the proposed techniques are essential to guide the process and assure a representative and sound experimental process.

Future Work

The work developed in this thesis opened new research directions and issues for future work, which can be divided into *short-* and *medium-term* goals. As short-term work we propose:

- **Validating the observations and conclusions obtained by assessing the performance of the models in a real-world production system** – validating that the results obtained in the various experimental evaluations hold on a production environment is a critical step towards promoting the investment and development of OFP solutions. Notwithstanding, this is not a trivial task, as it requires establishing a cooperation with a company/business where the failure predictors can be developed, deployed, and monitored.
- **Researching concept drift, batch, and online learning to detect and handle changes to the underlying problem to keep the models updated** – due to the dynamicity of modern systems it is common that their use, and underlying problem, changes over time. It is an active open issue and thus several works have tried to address this in other domains (e.g., [Malialis et al., 2020]). To widen the applicability of OFP it is highly relevant to study the best alternatives to detect such changes in the data (including new failure modes) and how to modify the systems to avoid losing performance.
- **Exploring state-of-the-art time series and deep learning to lever-**

age autocorrelation to increase the predictive performance – OFP is in itself a time series problem and thus there is autocorrelation between sequential values. Although this work acknowledges that, it did not thoroughly explore it. Deep learning algorithms have been successfully used in several domains with sequence/spatial correlation (e.g., [Lim and Zohren, 2021]). Similar approaches could be used to increase predictive performance and understand the evolution of failures by considering the sequential nature of the data.

- **Using white-box explainable ML algorithms to infer and determine the causes of failure** – understanding why ML models make certain predictions is extremely relevant and an expanding field of research (e.g., [Roscher et al., 2020]). Besides being able to predict failures, it is also important to understand the indicators and synergies that precede failures. This may allow identifying which (and why) components play a part in different failure modes, and such information may eventually be leveraged when creating the predictors or taking preemptive measures.
- **Using generative ML techniques as alternative approaches to generate failure data** – generative ML has been recurrently used in other domains, often as a means to generate synthetic realistic data [Creswell et al., 2018]. Although fault injection is the current best solution to generate failure data to support OFP, it still presents several limitations. The use of generative approaches may be a viable direction to replace or complement fault injection.

The medium-/long-term research directions envisioned are as follows:

- **Improving the robustness of failure predictors to variations in the data inherent to real systems** – as extensively shown in the literature, ML models can be sensitive even to minor variations in the data. Although the work in this thesis identifies this issue and the need to be aware and deal with it, achieving this in a representative and consistent way is still an open issue [Goodfellow et al., 2018].
- **Assessing the viability of using transfer learning between different OSs and workloads** – the ability to transfer knowledge from a task that has already been learned to another has been an active field of research [Zhuang et al., 2020]. The current assumption in OFP is that failures of distinct workloads and OSs will likely be different, and specific. However, it is also plausible that certain failures will share similar symptoms (e.g., memory leak). Researching such similarities between OSs and workloads could enable the use of transfer learning techniques across different scenarios.
- **Combining multiple sources of data to increase predictive performance** – the work conducted in this thesis focused exclusively on the use of system metrics to predict failures. However, an interesting research direction used in other domains is combining heterogeneous sources of data to increase performance (e.g., [Choi et al., 2020]). For OFP, combining multiple sources of data (e.g., metrics, logs) may allow detecting new symptoms

or improve existing predictors.

- **Implementing evolutionary approaches for optimizing hyperparameters and evolving solutions** – due to the variety of techniques and parameters of both OFP and ML, identifying the best set is a complex task. Moreover, in certain domains (e.g., deep learning) efficiently creating/defining the model is still an open issue. Evolutionary computation has been used to solve complex optimization problems in other domains by evolving potential solutions (e.g., [Al-Sahaf et al., 2019]). Besides addressing the search for techniques and parameters this also allows evolving, instead of designing, the solutions.
- **Expanding and assessing the use of OFP on cloud and distributed systems** – many modern complex infrastructures use cloud or distributed architectures. Although such systems are fault-tolerant by design they are still prone to failure (e.g., [Prathiba and Sowvarnica, 2017]). OFP could play a relevant role in predicting incoming failures, both at the node and system level.
- **Predicting the failure-proneness of the system** – instead of providing a categorical failure prediction, using regression models to predict the failure-proneness of the current state (e.g., between 0 and 100%) could provide a more progressive and detailed perspective of the expected reliability of the system (e.g., [Sun et al., 2012]).

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. `http://tensorflow.org/`. Accessed 2019-04-20.

Aggarwal, C. C. (2013). *Outlier Analysis*. Springer.

Ahmed, N. K., Atiya, A. F., El Gayar, N., and El-Shishiny, H. (2010). An empirical comparison of machine learning models for time series forecasting. *Econometric Reviews*, 29(5):594–621.

Al-Sahaf, H., Bi, Y., Chen, Q., Lensen, A., Mei, Y., Sun, Y., Tran, B., Xue, B., and Zhang, M. (2019). A survey on evolutionary machine learning. *Journal of the Royal Society of New Zealand*, 49(2):205–228.

Alpaydin, E. (2014). *Introduction to Machine Learning, 3rd ed., ser. Adaptive Computation and Machine Learning*. The MIT Press.

Alsina, E. F., Chica, M., Trawiński, K., and Regattieri, A. (2018). On the use of machine learning methods to predict component reliability from data-driven industrial case studies. *The International Journal of Advanced Manufacturing Technology*, 94(5):2419–2433.

Alves, H., Fonseca, B., and Antunes, N. (2016). Experimenting machine learning techniques to predict vulnerabilities. *Proceedings - 7th Latin-American Symposium on Dependable Computing, LADC 2016*, pages 151–156.

Amr, T. (2012). Survey on time-series data classification. *TSDM journal*.

Andrews, J. H., Briand, L. C., and Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? *Proceedings of the 27th international conference on Software engineering - ICSE '05*, page 402.

Antunes, N. and Vieira, M. (2010). Benchmarking vulnerability detection tools for web services. In *2010 IEEE International Conference on Web Services*, pages 203–210. IEEE.

Antunes, N. and Vieira, M. (2015). On the metrics for benchmarking vulnerability detection tools. In *2015 45th Annual IEEE/IFIP international conference on dependable systems and networks*, pages 505–516. IEEE.

Arık, S. O. and Pfister, T. (2020). Tabnet: Attentive interpretable tabular learning. *arXiv*.

Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J. C., Laprie, J. C., Martins, E., and Powell, D. (1990). Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2):166–182.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.

Ayodele, T. O. (2010). Types of Machine Learning Algorithms. *New Advances in Machine Learning*, pages 19–49.

Ballet, V., Aigrain, J., Laugel, T., Frossard, P., Detyniecki, M., et al. (2019). Imperceptible adversarial attacks on tabular data. In *NeurIPS 2019 Workshop on Robust AI in Financial Services: Data, Fairness, Explainability, Trustworthiness and Privacy (Robust AI in FS 2019)*.

Banabic, R. and Candea, G. (2012). Fast black-box testing of system recovery code. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 281–294. ACM.

Barandas, M., Folgado, D., Fernandes, L., Santos, S., Abreu, M., Bota, P., Liu, H., Schultz, T., and Gamboa, H. (2020). Tsfel: Time series feature extraction library. *SoftwareX*, 11:100456.

Bergmeir, C. and Benítez, J. M. (2012). On the use of cross-validation for time series predictor evaluation. *Information Sciences*, 191:192–213.

Bergmeir, C., Hyndman, R. J., and Koo, B. (2018). A note on the validity of cross-validation for evaluating autoregressive time series prediction. *Computational Statistics & Data Analysis*, 120:70–83.

Bhat, K., van der Kouwe, E., Bos, H., and Giuffrida, C. (2021). Firestarter: Practical software crash recovery with targeted library-level fault injection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 363–375. IEEE.

Bian, S. and Wang, W. (2007). On diversity and accuracy of homogeneous and heterogeneous ensembles. *International Journal of Hybrid Intelligent Systems*, 4(2):103–128.

Biggio, B. and Roli, F. (2018). Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331.

Bisgaard, S. and Kulahci, M. (2011). *Time series analysis and forecasting by example*. John Wiley & Sons.

Bontempi, G., Ben Taieb, S., and Le Borgne, Y. A. (2013). Machine learning strategies for time series forecasting. *Lecture Notes in Business Information Processing*, 138 LNBIP:62–77.

Bordes, A., Ertekin, S., Weston, J., and Bottou, L. (2005). Fast Kernel Classifiers with Online and Active Learning. *Journal of Machine Learning Research*, 6(Sep):1579–1619.

Borra, S. and Di Ciaccio, A. (2010). Measuring the prediction error. A comparison of cross-validation, bootstrap and covariance penalty methods. *Computational Statistics and Data Analysis*, 54(12):2976–2989.

Bouguerra, M. S., Gainaru, A., and Cappello, F. (2013). Failure Prediction: What To Do with Unpredicted Failures? *28th IEEE International Parallel & Distributed Processing Symposium.*

Brendel, W., Rauber, J., and Bethge, M. (2018). Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Brown, B. (2019). Facebook's catastrophic blackout could cost $90 million in lost revenue. `https://www.ccn.com/facebooks-blackout-90-million-lost-revenue/`. Accessed 2021-03-01.

Bruha, I. and Franek, F. (1996). Comparison of Various Routines for Unknown Attribute Value Processing: the Covering Paradigm. *International Journal of Pattern Recognition and Artificial Intelligence*, 10(08):939–955.

Candido, J., Melo, L., and d'Amorim, M. (2017). Test suite parallelization in open-source projects: a study on its usage and impact. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 838–848. IEEE Press.

Carlini, N. and Wagner, D. (2018). Audio adversarial examples: Targeted attacks on speech-to-text. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 1–7. IEEE.

Cawley, G. C. and Talbot, N. L. (2010). On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation. *Journal of Machine Learning Research*, 11:2079–2107.

Cerveira, F., Barbosa, R., and Madeira, H. (2017). Experience report: On the impact of software faults in the privileged virtual machine. In *28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 136–145. IEEE.

Chakraborty, A., Alam, M., Dey, V., Chattopadhyay, A., and Mukhopadhyay, D. (2018). Adversarial attacks and defences: A survey.

Chatfield, C. (2016). *The analysis of time series: an introduction.* CRC press.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357.

Chen, H., Zhang, H., Boning, D., and Hsieh, C.-J. (2019a). Robust decision trees against adversarial examples. In *International Conference on Machine Learning*, pages 1122–1131. PMLR.

Chen, H., Zhang, H., Si, S., Li, Y., Boning, D., and Hsieh, C.-J. (2019b). Robustness verification of tree-based models. In *Advances in Neural Information Processing Systems*, pages 12317–12328.

Chen, J., Jordan, M. I., and Wainwright, M. J. (2020). Hopskipjumpattack: A query-efficient decision-based attack. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1277–1294. IEEE.

Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA. ACM.

Chillarege, R. (1996). Orthogonal defect classification. *Handbook of Software Reliability Engineering*, pages 359–399.

Choi, D., Sumner, S. A., Holland, K. M., Draper, J., Murphy, S., Bowen, D. A., Zwald, M., Wang, J., Law, R., Taylor, J., et al. (2020). Development of a machine learning model using multiple, heterogeneous data sources to estimate weekly us suicide fatalities. *JAMA network open*, 3(12):e2030932–e2030932.

Choi, E., Hewlett, D., Uszkoreit, J., Polosukhin, I., Lacoste, A., and Berant, J. (2017). Coarse-to-fine question answering for long documents. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 209–220.

Chollet, F. et al. (2015). Keras. `https://keras.io`. Accessed 2019-04-20.

Cook, D. (2016). *Practical machine learning with H2O: powerful, scalable techniques for deep learning and AI.* " O'Reilly Media, Inc.".

Costa, P., Silva, J. G., and Madeira, H. (2009). Dependability benchmarking using software faults: How to create practical and representative faultloads. *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2009*, pages 289–294.

Costa, P., Vieira, M., Madeira, H., and Silva, J. G. (2003). Plug and play fault injector for dependability benchmarking. In *Latin-American Symposium on Dependable Computing*, pages 8–22. Springer.

Costa, P. M. L. N. d. (2013). *Dependability Benchmarking for Large and Complex Systems.* PhD thesis, University of Coimbra, Portugal.

Costa, V. S., Farias, A. D. S., Bedregal, B., Santiago, R. H., and Canuto, A. M. d. P. (2018). Combining multiple algorithms in classifier ensembles using generalized mixture functions. *Neurocomputing*, 313:402–414.

Cotroneo, D., De Simone, L., Liguori, P., and Natella, R. (2020). Fault injection analytics: A novel approach to discover failure modes in cloud-computing systems. *IEEE Transactions on Dependable and Secure Computing*.

Cotroneo, D., De Simone, L., Liguori, P., Natella, R., and Bidokhti, N. (2019). How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 200–211.

Cotroneo, D., De Simone, L., and Natella, R. (2017). Nfv-bench: A dependability benchmark for network function virtualization systems. *IEEE Transactions on Network and Service Management*, 14(4):934–948.

Cotroneo, D., De Simone, L., and Natella, R. (2018). Run-time detection of protocol bugs in storage i/o device drivers. *IEEE Transactions on Reliability*, 67(3):847–869.

Cotroneo, D., Lanzaro, A., Natella, R., and Barbosa, R. (2012). Experimental Analysis of Binary-Level Software Fault Injection in Complex Software. *2012 Ninth European Dependable Computing Conference*, pages 162–172.

Cotroneo, D., Simone, L. D., Fucci, F., and Natella, R. (2015). MoIO: Run-Time Monitoring for I/O Protocol Violations in Storage Device Drivers. *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 472–483.

Cover, T. M. and Thomas, J. A. (2012). *Elements of information theory*. John Wiley & Sons.

Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., and Bharath, A. A. (2018). Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65.

Crouzet, Y. and Kanoun, K. (2012). System dependability: Characterization and benchmarking. In *Advances in Computers*, volume 84, pages 93–139. Elsevier.

Daran, M. and Thevenod-Fosse, P. (1996). Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. *Issta*, pages 158–171.

DataRobot, I. (n.d.). Datarobot. `https://datarobot.com/`. Accessed 2019-04-20.

De Gooijer, J. G. and Hyndman, R. J. (2006). 25 Years of Time Series Forecasting. *International Journal of Forecasting*, 22(3):443–473.

de Sá, J. M. (2012). *Pattern Recognition: Concepts, Methods and Applications*. Springer Science & Business Media.

Depoutovitch, A. and Stumm, M. (2010). Otherworld: giving applications a chance to survive os kernel crashes. In *Proceedings of the 5th European conference on Computer systems*, pages 181–194.

Dhanalaxmi, B., Naidu, G. A., and Anuradha, K. (2015). A review on software fault detection and prevention mechanism in software development activities. *Journal of Computer Engineering*, 17(6):25–30.

Dietterich, T. (2002). Machine learning for sequential data: A review. *Structural, syntactic, and statistical pattern recognition*, pages 1–15.

Duboue, P. (2020). *The Art of Feature Engineering: Essentials for Machine Learning*. Cambridge University Press.

Durães, J. and Madeira, H. (2006). Emulation of software faults: A field data study and a practical approach. *IEEE transactions on software engineering*, 32(11):849–867.

Durães, J., Vieira, M., and Madeira, H. (2004). Dependability benchmarking of web-servers. In *International Conference on Computer Safety, Reliability, and Security*, pages 297–310. Springer.

Ed Targett, C. B. R. (2018). Amazon outage: Estimated $99 million lost. `https://www.cbronline.com/news/amazon-outage-lost-sales`. Accessed 2021-03-01.

Eibe, F., Hall, M., and Witten, I. (2016). The weka workbench. practical machine learning tools and techniques. *Morgan Kaufmann*.

Elliott, A. (2019). *The culture of AI: Everyday life and the digital revolution*. Routledge.

Eshete, B. and Venkatakrishnan, V. N. (2017). DynaMiner: Leveraging Offline Infection Analytics for On-the-Wire Malware Detection. *Proceedings - 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017*, pages 463–474.

Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xiao, C., Prakash, A., Kohno, T., and Song, D. (2018). Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1625–1634.

Feedzai, I. (n.d.). Feedzai ai. `https://feedzai.com/`. Accessed 2019-04-20.

Fernández, J. and Dullaert, W. (2018). D3.3 - Scalability and Robustness testing report V1. `https://doi.org/10.5281/zenodo.2545149`.

Field, A. (2013). *Discovering Statistics Using IBM SPSS Statistics*. Sage Publications Ltd., 4th edition.

Fleiss, J. L. (1981). *Statistical methods for rates and proportions*. John Wiley & Sons.

Friedman, J. H. (1997). Data mining and statistics: what's the connection? In *Keynote Address, 29th Symposium on the Interface: Computing Science and Statistics*.

Gabor, U. T., Siegert, D. F., and Spinczyk, O. (2019). High-accuracy software fault injection in source code with clang. In *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 75–7509. IEEE.

Gambi, A., Kappler, S., Lampel, J., and Zeller, A. (2017). Cut: automatic unit testing in the cloud. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 364–367. ACM.

Gan, H., Sang, N., Huang, R., Tong, X., and Dan, Z. (2013). Using clustering analysis to improve semi-supervised classification. *Neurocomputing*, 101:290–298.

Gao, X. (2018). Deep reinforcement learning for time series: playing idealized trading games.

Gavrilov, A. (2018). Yet another expect for java. `https://github.com/Alexey1Gavrilov/ExpectIt`. Accessed 2019-11-15.

Ghosh, A. (2004). *Evolutionary computation in data mining*, volume 163. Springer Science & Business Media.

GitLab (n.d.). Gitlab continuous integration (ci) / continuous delivery (cd). `https://about.gitlab.com/product/continuous-integration/`. Accessed 2019-11-12.

Globerson, A. and Roweis, S. (2006). Nightmare at test time: robust learning by feature deletion. In *Proceedings of the 23rd international conference on Machine learning*, pages 353–360.

Goodfellow, I., McDaniel, P., and Papernot, N. (2018). Making machine learning robust against adversarial inputs. *Communications of the ACM*, 61(7):56–66.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015). Explaining and harnessing adversarial examples. In Bengio, Y. and LeCun, Y., editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Gray, J. (1986). Why Do Computers Stop and What Can Be Done About It? *Technical Report - Tandem Computers*, 3(June):3–12.

Gray, J. (1992). *Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Grimm, R. (2017). Pros and cons of the various memory allocation strategies. https://www.modernescpp.com/index.php/pros-and-cons-of-the-various-memory-management-strategies. Accessed 2019-11-15.

Grzymala-Busse, J. W. and Hu, M. (2001). A Comparison of Several Approaches to Missing Attribute Values in Data Mining BT - Rough sets and current trends in computing. *Rough sets and current trends in computing*, 2005(Chapter 46):378–385.

Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. (2006). Enforcing performance isolation across virtual machines in xen. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 342–362. Springer.

H2O.ai (n.d.). H2o.ai git repository. https://github.com/h2oai/h2o-3. Accessed 2019-04-20.

Hajdu, Á., Ivaki, N., Kocsis, I., Klenik, A., Gönczy, L., Laranjeiro, N., Madeira, H., and Pataricza, A. (2020). Using fault injection to assess blockchain systems in presence of faulty smart contracts. *IEEE Access*, 8:190760–190783.

Hat, R. (n.d.a). Interrupts and irq tuning. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-irq. Accessed 2019-11-15.

Hat, R. (n.d.b). Offloading rcu callbacks. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/offloading_rcu_callbacks. Accessed 2019-11-15.

Hat, R. (n.d.c). What is kvm? https://www.redhat.com/en/topics/virtualization/what-is-KVM. Accessed 2021-06-15.

Heaton, J. (2015). *Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks*. CreateSpace Independent Publishing Platform, 1st edition.

Herff, C. and Krusienski, D. J. (2019). Extracting features from time series. *Fundamentals of Clinical Data Science*, pages 85–100.

Hetzel, W. C. and Hetzel, B. (1988). *The complete guide to software testing*. QED Information Sciences Wellesley, MA.

Hoffmann, G., Trivedi, K., and Malek, M. (2006). A Best Practice Guide to Resources Forecasting for the Apache Webserver. *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 183–193.

Hogg, R. V. and Tanis, E. A. (2009). *Probability and statistical inference*. Pearson Educational International.

Holcomb, S. D., Porter, W. K., Ault, S. V., Mao, G., and Wang, J. (2018). Overview on deepmind and its alphago zero ai. In *Proceedings of the 2018 international conference on big data and education*, pages 67–71.

Hosseini, M.-P., Lu, S., Kamaraj, K., Slowikowski, A., and Venkatesh, H. C. (2020). Deep learning architectures. In *Deep learning: concepts and architectures*, pages 1–24. Springer.

Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. (1997). Fault injection techniques and tools. *Computer*, 30(April):75–82.

Hughes, G. F., Murray, J. F., Kreutz-Delgado, K., and Elkan, C. (2002). Improved disk-drive failure warnings. *IEEE transactions on reliability*, 51(3):350–357.

IBM (n.d.). Hypervisors. `https://www.ibm.com/cloud/learn/hypervisors`. Accessed 2021-06-15.

Intel (n.d.a). Intel® hyper-threading technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`. Accessed 2019-11-06.

Intel (n.d.b). Intel® turbo boost technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`. Accessed 2019-11-06.

Intel (n.d.c). Intel® vt technology. `https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html`. Accessed 2019-11-06.

Irrera, I. (2016). *Fault Injection for Online Failure Prediction Assessment and Improvement*. PhD thesis, University of Coimbra, Portugal.

Irrera, I., Durães, J., Madeira, H., and Vieira, M. (2013a). Assessing the impact of virtualization on the generation of failure prediction data. *Proceedings - 6th Latin-American Symposium on Dependable Computing, LADC 2013*, pages 92–97.

Irrera, I., Durães, J., and Vieira, M. (2014). On the Need for Training Failure Prediction Algorithms in Evolving Software Systems. *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 216–223.

Irrera, I., Pereira, C., and Vieira, M. (2013b). The time dimension in predicting failures: A case study. *Proceedings - 6th Latin-American Symposium on Dependable Computing, LADC 2013*, pages 86–91.

Irrera, I. and Vieira, M. (2014). A practical approach for generating failure data for assessing and comparing failure prediction algorithms. In *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, pages 86–95. IEEE.

Irrera, I. and Vieira, M. (2015). Towards assessing representativeness of fault injection-generated failure data for online failure prediction. In *2015 IEEE International Conference on Dependable Systems and Networks Workshops*, pages 75–80. IEEE.

Irrera, I., Vieira, M., and Durães, J. (2015). Adaptive Failure Prediction for Computer Systems: A Framework and a Case Study. *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pages 142–149.

Irrera, I., Zentai, A., Cunha, J. C., and Madeira, H. (2017). Validating a safety critical railway application using fault injection. *Certifications of Critical Systems-The CECRIS Experience*, page 227.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning*, volume 103 of *Springer Texts in Statistics*. Springer New York, New York, NY.

Jassas, M. and Mahmoud, Q. H. (2018). Failure analysis and characterization of scheduling jobs in google cluster trace. In *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*, pages 3102–3107. IEEE.

Jassas, M. S. and Mahmoud, Q. H. (2020). Evaluation of a failure prediction model for large scale cloud applications. In *Canadian Conference on Artificial Intelligence*, pages 321–327. Springer.

Jeong, E., Lee, N., Kim, J., Kang, D., and Ha, S. (2017). Fifa: A kernel-level fault injection framework for arm-based embedded linux system. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 23–34. IEEE.

Jiawei Han, Micheline Kamber, J. P. (2011). *Data Mining: Concepts and Techniques*. Elsevier, 3rd edition.

Jing, W., Guan, N., and Yi, W. (2014). Performance isolation for real-time systems with xen hypervisor on multi-cores. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–7. IEEE.

Jones, E., Oliphant, T., Peterson, P., et al. (2001–). SciPy: Open source scientific tools for Python. `http://www.scipy.org/`. Accessed 2019-04-20.

Jordan, P. L., Peterson, G. L., Lin, A. C., Mendenhall, M. J., and Sellers, A. J. (2017). Narrowing the scope of failure prediction using targeted fault load injection. *Enterprise Information Systems*, 00(00):1–16.

Juristo, N. and Moreno, A. M. (2013). *Basics of software engineering experimentation*. Springer Science & Business Media.

Kanoun, K. and Spainhower, L. (2008). *Dependability benchmarking for computer systems*, volume 72. Wiley Online Library.

Kantchelian, A., Tygar, J. D., and Joseph, A. (2016). Evasion and hardening of tree ensemble classifiers. In *International Conference on Machine Learning*, pages 2387–2396.

Kapfhammer, G. M. (2001). Automatically and transparently distributing the execution of regression test suites. In *Proceedings of the 18th International Conference on Testing Computer Software*, volume 18.

Kelleher, J. D., Namee, B. M., and D'Arcy, A. (2015). *Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies*. The MIT Press.

Keniston, J., Panchamukhi, P. S., and Hiramatsu, M. (n.d.). Kernel probes (kprobes). `https://www.kernel.org/doc/Documentation/kprobes.txt`. Accessed 2020-05-20.

Kernel, L. (n.d.). Kernel samepage merging (ksm). `https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html`. Accessed 2021-09-16.

Kikuchi, N., Yoshimura, T., Sakuma, R., and Kono, K. (2014). Do injected faults cause real failures? a case study of linux. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 174–179. IEEE.

Kim, J. and Kim, J. (2018). The impact of imbalanced training data on machine learning for author name disambiguation. *Scientometrics*, 117(1):511–526.

Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Al Sallab, A. A., Yogamani, S., and Pérez, P. (2021). Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*.

Koopman, P., Sung, J., Dingman, C., Siewiorek, D., and Marz, T. (1997). Comparing operating systems using robustness benchmarks. In *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on*, pages 72–79.

Kotsiantis, S. B., Kanellopoulos, D., and Pintelas, P. E. (2006). Data preprocessing for supervised learning. *International Journal of Computer Science*, 1(2):111–117.

Kouwe, E. V. D. and Tanenbaum, A. S. (2016). HSFI: Accurate fault injection scalable to large code bases. *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*, pages 144–155.

Kropp, N. P., Koopman, P. J., and Siewiorek, D. P. (1998). Automated robustness testing of off-the-shelf software components. *Digest of Papers - 28th Annual International Symposium on Fault-Tolerant Computing, FTCS 1998*, 1998-January:1–10.

Kuhn, R., Kacker, R., Lei, Y., and Hunter, J. (2009). Combinatorial software testing. *Computer*, 42(8):94–96.

Kuncheva, L. I. and Whitaker, C. J. (2003). Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. *Machine learning*, 51(2):181–207.

Kuznetsova, A., Hwang, S. J., Rosenhahn, B., and Sigal, L. (2015). Expanding object detector's horizon: Incremental learning framework for object detection in videos. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 28–36.

KVM (n.d.). Kvm. `https://www.linux-kvm.org/page/Main_Page`. Accessed 2019-11-15.

Kwon, Y., Dunn, A. M., Lee, M. Z., Hofmann, O. S., Xu, Y., and Witchel, E. (2016). Sego: Pervasive trusted metadata for efficiently verified untrusted system services. *ACM SIGARCH Computer Architecture News*, 44(2):277–290.

Laboratory, L. A. N. (n.d.). Usrc data sources failure data. `https://www.lanl.gov/projects/ultrascale-systems-research-center/data/failure-data.php`. Accessed 2021-03-01.

Lakshminarayan, K., Harp, S. A., and Samad, T. (1999). Imputation of missing data in industrial databases. *Applied Intelligence*, 11(3):259–275.

Lameter, C. (n.d.). Numa (non-uniform memory access): An overview. `https://queue.acm.org/detail.cfm?id=2513149`. Accessed 2019-11-15.

Legg, S. and Hutter, M. (2006). A formal definition of intelligence for artificial systems. *Proc. 50th Anniversary Summit of Artificial Intelligence*, pages 197–198.

Lemaître, G., Nogueira, F., and Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research*, 18(1):559–563.

Libes, D. (1995). *Exploring Expect: A Tcl-based toolkit for automating interactive programs.* " O'Reilly Media, Inc.".

Libes, D. (n.d.). Expect linux command/library. `https://linux.die.net/man/1/expect`. Accessed 2019-11-15.

Lim, B. and Zohren, S. (2021). Time-series forecasting with deep learning: a survey. *Philosophical Transactions of the Royal Society A*, 379(2194):20200209.

Liu, A. Y.-c. (2004). *The effect of oversampling and undersampling on classifying imbalanced text datasets.* PhD thesis, University of Texas.

Liu, H. and Motoda, H. (2013). *Instance selection and construction for data mining*, volume 608. Springer Science & Business Media.

Lu, Q., Farahani, M., Wei, J., Thomas, A., and Pattabiraman, K. (2015). Llfi: An intermediate code-level fault injection tool for hardware faults. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 11–16. IEEE.

LWN.net (n.d.). (nearly) full tickless operation in 3.10. `https://lwn.net/Articles/549580/`. Accessed 2019-11-15.

Makai, J., Peters, A. J., Bitzes, G., Sindrilaru, E. A., Simon, M. K., and Manzi, A. (2019). Testing of complex, large-scale distributed storage systems: a cern disk storage case study. In *EPJ Web of Conferences*, volume 214, page 05008. EDP Sciences.

Malialis, K., Panayiotou, C. G., and Polycarpou, M. M. (2020). Online learning with adaptive rebalancing in nonstationary environments. *IEEE Transactions on Neural Networks and Learning Systems*.

Marsland, S. (2014). Machine learning: An algorithmic perspective.

Martínez, M., de Andrés, D., Ruiz, J. C., and Friginal, J. (2014). From measures to conclusions using analytic hierarchy process in dependability benchmarking. *IEEE Trans. Instrum. Meas.*, 63(11):2548–2556.

Matthews, J. N., Hu, W., Hapuarachchi, M., Deshane, T., Dimatos, D., Hamilton, G., McCabe, M., and Owens, J. (2007). Quantifying the performance isolation properties of virtualization systems. In *2007 workshop on Experimental computer science*, page 6. ACM.

McCalpin, J. D. (1995). Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25.

McCarthy, J. (2007). What is Artificial Intelligence ?

McFall-Johnsen, M. (2020). Catastrophic software errors doomed boeing's airplanes. `https://www.businessinsider.com/boeing-software-errors-jeopardized-starliner-spaceship-737-max-planes-2020-2`. Accessed 2021-03-01.

Meng, D. and Chen, H. (2017). Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 135–147.

Munin (n.d.). Munin. `http://munin-monitoring.org/`. Accessed 2019-11-15.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.

Nagios (n.d.). Nagios. `https://www.nagios.com/`. Accessed 2019-11-15.

Natella, R. and Cotroneo, D. (2010). Emulation of transient software faults for dependability assessment: A case study. *EDCC-8 - Proceedings of the 8th European Dependable Computing Conference*, pages 23–32.

Natella, R., Cotroneo, D., Durães, J., and Madeira, H. (2010). Representativeness analysis of injected software faults in complex software. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 437–446.

Natella, R., Cotroneo, D., Durães, J., and Madeira, H. S. (2012). On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96.

Natella, R., Cotroneo, D., and Madeira, H. S. (2016). Assessing Dependability with Software Fault Injection. *ACM Computing Surveys*, 48(3):1–55.

Netdata (n.d.). Netdata. `https://www.netdata.cloud/`. Accessed 2019-11-15.

Ng, A. (2017). Machine learning yearning. 2018.

Ng, W. T. and Chen, P. M. (1999). The systematic improvement of fault tolerance in the rio file cache. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*, pages 76–83. IEEE.

Nie, B., Xue, J., Gupta, S., Patel, T., Engelmann, C., Smirni, E., and Tiwari, D. (2018). Machine learning models for gpu error prediction in a large scale hpc system. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 95–106. IEEE.

Novaković, D., Vasić, N., Novaković, S., Kostić, D., and Bianchini, R. (2013). Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 219–230.

Novikov, A. (2018). pyclustering. `https://doi.org/10.5281/zenodo.1254845`. Accessed 2019-04-20.

Nunes, P., Medeiros, I., Fonseca, J. C., Neves, N., Correia, M., and Vieira, M. (2018). Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175.

Oppenheimer, D., Patterson, D., and Ganapathi, A. (2003). Why do Internet Services fail, and what can be done about it? *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems-Volume 4*, pages 1–1.

Oracle (n.d.). Oracle virtualbox. `https://www.virtualbox.org/`. Accessed 2019-11-15.

Oriol, M. and Ullah, F. (2010). Yeti on the cloud. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 434–437. IEEE.

Palit, A. K. and Popovic, D. (2006). *Computational intelligence in time series forecasting: theory and engineering applications*. Springer Science & Business Media.

Palit, H. N., Li, X., Lu, S., Larsen, L. C., and Setia, J. A. (2013). Evaluating hardware-assisted virtualization for deploying hpc-as-a-service. In *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*, pages 11–20.

Papernot, N., Mcdaniel, P., and Goodfellow, I. (2016). Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *ArXiv*, abs/1605.07277.

Parr, T., Turgutlu, K., Csiszar, C., and Howard, J. (n.d.). Beware default random forest importances. `https://explained.ai/rf-importance/`. Accessed 2020-05-20.

Parveen, T., Tilley, S., Daley, N., and Morales, P. (2009). Towards a distributed execution framework for junit test cases. In *2009 IEEE International Conference on Software Maintenance*, pages 425–428. IEEE.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Pitakrat, T., Grunert, J., Kabierschke, O., Keller, F., and Van Hoorn, A. (2014). A framework for system event classification and prediction by means of machine learning. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*, pages 173–180.

Pitakrat, T., Okanović, D., van Hoorn, A., and Grunske, L. (2018). Hora: Architecture-aware online failure prediction. *Journal of Systems and Software*, 137:669–685.

Polikar, R. (2006). Ensemble based systems in decision making. *Circuits and Systems Magazine, IEEE*, 6(3):21–45.

Powell, D. et al. (2001). *A generic fault-tolerant architecture for real-time dependable systems*, volume 242. Springer.

Prathiba, S. and Sowvarnica, S. (2017). Survey of failures and fault tolerance in cloud. In *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)*, pages 169–172. IEEE.

QEMU (n.d.). Qemu. `https://www.qemu.org/`. Accessed 2019-11-15.

Rácz, A., Bajusz, D., and Héberger, K. (2021). Effect of dataset size and train/test split ratios in qsar/qspr multiclass classification. *Molecules*, 26(4):1111.

Ren, Y., Lin, J., Tang, S., Zhou, J., Yang, S., Qi, Y., and Ren, X. (2020). Generating natural language adversarial examples on a large scale with generative models. In Giacomo, G. D., Catalá, A., Dilkina, B., Milano, M., Barro, S., Bugarín, A., and Lang, J., editors, *ECAI 2020 - 24th European Conference on*

Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020), volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2156–2163. IOS Press.

Roscher, R., Bohn, B., Duarte, M. F., and Garcke, J. (2020). Explainable machine learning for scientific insights and discoveries. *Ieee Access*, 8:42200–42216.

Russell, S. and Norvig, P. (2021). *Artificial Intelligence: A Modern Approach, Global Edition*. Pearson, 4 edition.

Saez, Y., Baldominos, A., and Isasi, P. (2017). A comparison study of classifier algorithms for cross-person physical activity recognition. *Sensors (Switzerland)*, 17(1).

Salfner, F., Hoffmann, G. A., and Malek, M. (2005). Prediction-based software availability enhancement. In Babaoglu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A. P. A., and van Steen, M., editors, *Self-star Properties in Complex Information Systems, Conceptual and Practical Foundations*, volume 3460 of *Lecture Notes in Computer Science*, pages 143–157. Springer.

Salfner, F., Lenk, M., and Malek, M. (2010). A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42(3):10:1–10:42.

Salfner, F. and Malek, M. (2007). Using hidden semi-markov models for effective online failure prediction. In *26th IEEE Symposium on Reliable Distributed Systems (SRDS 2007), Beijing, China, October 10-12, 2007*, pages 161–174. IEEE Computer Society.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229.

Santos, T. and Kern, R. (2016). A literature survey of early time series classification and deep learning. In Kern, R., Reiner, G., and Bluder, O., editors, *Proceedings of the 1st International Workshop on Science, Application and Methods in Industry 4.0 co-located with (i-KNOW 2016), Graz, Austria, October 19, 2016*, volume 1793 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Sauvanaud, C., Lazri, K., Kaâniche, M., and Kanoun, K. (2016). Anomaly detection and root cause localization in virtual network functions. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 196–206. IEEE.

Schwahn, O., Coppik, N., Winter, S., and Suri, N. (2019a). Assessing the state and improving the art of parallel testing for c. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 123–133. ACM.

Schwahn, O., Coppik, N., Winter, S., and Suri, N. (2019b). FastFI: Accelerating software fault injections. *Proceedings of IEEE Pacific Rim International Symposium on Dependable Computing, PRDC*, 2018-Decem:193–202.

Sesmero, M. P., Ledezma, A. I., and Sanchis, A. (2015). Generating ensembles of heterogeneous classifiers using Stacked Generalization. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 5(1):21–34.

Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.

Sharif, M., Bauer, L., and Reiter, M. K. (2018). On the suitability of lp-norms for creating and preventing adversarial examples. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1605–1613.

Siegel, E. (2016). *Predictive analytics: The power to predict who will click, buy, lie, or die*. Wiley Hoboken.

Siewiorek, D. P. and Swarz, R. S. (1998). *Reliable computer systems: design and evaluation*. AK Peters/CRC Press.

Silipo, R., Adae, I., Hart, A., and Berthold, M. (2014). Seven Techniques for Dimensionality Reduction. `https://www.knime.com/sites/default/files/inline-images/knime_seventechniquesdatadimreduction.pdf`.

Smith, M. R., Martinez, T., and Giraud-Carrier, C. (2014). An instance level analysis of data complexity. *Machine learning*, 95(2):225–256.

Sokolova, M. and Lapalme, G. (2009). A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437.

Song, Y., Shu, R., Kushman, N., and Ermon, S. (2018). Constructing unrestricted adversarial examples with generative models. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Spurrier, N. (n.d.). Pexpect. `https://pexpect.readthedocs.io/en/stable/`. Accessed 2019-11-15.

Strobl, C., Boulesteix, A.-L., Zeileis, A., and Hothorn, T. (2007). Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC bioinformatics*, 8(1):25.

Sun, J., Zuo, H., Wang, W., and Pecht, M. G. (2012). Application of a state space modeling technique to system prognostics based on a health index for condition-based maintenance. *Mechanical systems and signal processing*, 28:585–596.

Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M. (2006). Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4):333–360.

sysstat (n.d.). sysstat - collection of performance monitoring tools for linux. http://sebastien.godard.pagesperso-orange.fr/documentation.html. Accessed 2019-11-15.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., and Fergus, R. (2014). Intriguing properties of neural networks. In Bengio, Y. and LeCun, Y., editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

Tan, P.-N., Steinbach, M., and Kumar, V. (2005). *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Tashman, L. J. (2000). Out-of-sample tests of forecasting accuracy: an analysis and review. *International journal of forecasting*, 16(4):437–450.

Tiao, G. C. and Tsay, R. S. (1994). Some advances in non-linear and adaptive modelling in time-series. *Journal of Forecasting*, 13(2):109–131.

Ting, K. M. and Witten, I. H. (1999). Issues in stacked generalization. *Journal of artificial intelligence research*, 10:271–289.

Tipton, H. F. and Krause, M. (2007). *Information security management handbook*. CRC press.

Turney, P. D. (2002). Types of cost in inductive concept learning. *CoRR*, cs.LG/0212034.

Ubuntu (n.d.a). chrt - manipulate the real-time attributes of a process. http://manpages.ubuntu.com/manpages/trusty/man1/chrt.1.html. Accessed 2019-11-15.

Ubuntu (n.d.b). cset-proc. http://manpages.ubuntu.com/manpages/bionic/man1/cset-proc.1.html. Accessed 2019-11-15.

Ubuntu (n.d.c). cset-set. http://manpages.ubuntu.com/manpages/bionic/man1/cset-set.1.html. Accessed 2019-11-15.

Ubuntu (n.d.d). cset-shield. http://manpages.ubuntu.com/manpages/trusty/man1/cset-shield.1.html. Accessed 2019-11-15.

Ubuntu (n.d.e). Real-time/low-latency kernel. https://help.ubuntu.com/community/UbuntuStudio/RealTimeKernel. Accessed 2019-11-15.

Ubuntu (n.d.f). stress-ng. https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html. Accessed 2020-10-15.

Usenix and University, C. M. (n.d.). Computer failure data repository. https://www.usenix.org/cfdr. Accessed 2020-10-15.

Van Der Kouwe, E., Giuffrida, C., and Tanenbaum, A. S. (2014). Evaluating distortion in fault injection experiments. *Proceedings - 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering, HASE 2014*, pages 25–32.

Van Der Kouwe, E. and Tanenbaum, A. S. (2016). Hsfi: Accurate fault injection scalable to large code bases. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 144–155. IEEE.

Vieira, M. and Madeira, H. (2003). A dependability benchmark for OLTP application environments. *Proceedings of the 29th international conference on Very large data bases - Volume 29*, 94:742–753.

Vieira, M. and Madeira, H. (2005). Towards a security benchmark for database management systems. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 592–601.

Vilar, J. M. (2009). Classifying Time Series Data : A Nonparametric Approach. *Journal of Classification*, 8(April):3–28.

VMWare (n.d.a). Esxi. `https://www.vmware.com/products/esxi-and-esx.html`. Accessed 2021-06-15.

VMWare (n.d.b). Vmware workstation pro. `https://www.vmware.com/products/workstation-pro.html`. Accessed 2021-06-15.

Webb, G. I., Hyde, R., Cao, H., Nguyen, H. L., and Petitjean, F. (2016). Characterizing concept drift. *Data Mining and Knowledge Discovery*, 30(4):964–994.

Wild, P., Radu, P., Chen, L., and Ferryman, J. (2016). Robust multimodal face and fingerprint fusion in the presence of spoofing attacks. *Pattern Recognition*, 50:17–25.

Winter, S., Schwahn, O., Natella, R., Suri, N., and Cotroneo, D. (2015). No PAIN, no gain? The utility of parallel fault injections. *Proceedings - International Conference on Software Engineering*, 1:494–505.

Winter, S., Tretter, M., Sattler, B., and Suri, N. (2013). simfi: From single to simultaneous software fault injections. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE.

Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. (2016). *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition.

Wolpert, D. H. (1994). The Relationship between PAC, the Statistical Physics framework, the Bayesian framework, and the VC framework. *The Mathematics of Generalization*, pages 1–96.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

Wolski, R., Spring, N., and Peterson, C. (1997). Implementing a performance forecasting system for metacomputing. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '97*, pages 1–19, New York, New York, USA. ACM Press.

Wu, L., Zhu, Z., Tai, C., and E, W. (2018). Understanding and enhancing the transferability of adversarial examples.

Xen (n.d.). Xen project. `https://xenproject.org/`. Accessed 2019-11-15.

Xu, H., Caramanis, C., and Mannor, S. (2009). Robustness and regularization of support vector machines. *Journal of machine learning research*, 10(7).

Yoshimura, T., Yamada, H., and Kono, K. (2012). Is Linux Kernel Oops Useful or Not? *HotDep*.

Yoshimura, T., Yamada, H., and Kono, K. (2013). Using Fault Injection to Analyze the Scope of Error Propagation in Linux. *IPSJ Online Transactions*, 6(2):55–64.

Zhang, G. (2003). Time series forecasting using a hybrid ARIMA and neural network model. *Neurocomputing*, 50:159–175.

Zhang, J., Zhou, K., Huang, P., He, X., Xie, M., Cheng, B., Ji, Y., and hu Wang, Y. (2020). Minority disk failure prediction based on transfer learning in large data centers of heterogeneous disk systems. *IEEE Transactions on Parallel and Distributed Systems*.

Zhao, L., Liu, T., Peng, X., and Metaxas, D. N. (2020). Maximum-entropy adversarial data augmentation for improved generalization and robustness. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Zhao, Z., Dua, D., and Singh, S. (2018). Generating natural adversarial examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Zheng, S., Song, Y., Leung, T., and Goodfellow, I. (2016). Improving the robustness of deep neural networks via stability training. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4480–4488.

Zhou, Z.-H. (2012). *Ensemble methods: foundations and algorithms*. CRC press.

Zhu, B., Wang, G., Liu, X., Hu, D., Lin, S., and Ma, J. (2013). Proactive drive failure prediction for large scale storage systems. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*, pages 1–5. IEEE.

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., and He, Q. (2020). A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76.

Appendixes

Appendix A: OFP on Windows OS

The experimental evaluation presented in this thesis demonstrated that it is possible to develop accurate failure predictors for the Linux OS. To validate if these observations could be generalized, an exploratory study was also conducted on an existing failure dataset targeting Windows XP. The goal of this analysis is to provide assurances and external validity concerning the ability to create accurate failure predictors regardless of the fault injector or OS. All the following experiments were conducted using Propheticus.

A.1 Dataset

The data used in this study is entirely different from the one used in the previous section: different hypervisors, OS, fault injector, fault model, and workloads. It was generated to study the injection of realistic faults on Windows XP (SP3) using the G-SWFIT technique [Irrera et al., 2013a]. The failure modes considered are system *crash* and *hang*. It contains 233 numeric system variables. The experiments were run in three environments: a *real* machine (without virtualization), and two *virtualized* environments (i.e., VMWare vSphere and Citrix XEN). Once again, as not all experiments led to failures, the data used were limited to a maximum of 100000 samples per dataset. Three configurations were tested in this work for the pairs $\Delta t_l, \Delta t_p$: [20,20], [40,20], and [60,20], considering a 'short', 'medium', and 'long' term prediction.

As most features are based on different scales, a Z-score standardization was applied. Runs where the failure was observed immediately after fault activation were discarded. Both descriptive and exploratory analyses were conducted. Similar to the Linux dataset (that we created), by analyzing the mean values of the features per failure mode it was possible to observe that non-failure samples were noticeably different from the failure samples. On the other hand, the differences between the *hang* and *crash* samples are not so considerable, although for some features they are still differentiable.

A.2 ML Algorithms and Techniques

For this study, several different algorithms were considered:*SVM, DT, RF, Bagging, Adaboost, Extra Trees, NN* (a MLP once again) *Naive Bayes, Gaussian Process, Logistic Regression*, and *k Nearest Neighbors (k-NN)*. Both *filter* (by *variance*, and *correlation*) and *wrapper* (through *Recursive Feature Elimination (RFE)*) feature selection methods were considered. Features with zero variance and correlation >90% were removed. RFE used 10 DTs and a cross-validation approach was used to select which subset of features are kept. To study the effect of the imbalance in the data, a simple data undersampling method, *Random Undersampling*, was used.

For simplicity, most of the parameters of the various algorithms were chosen based on existing literature, and only the most relevant configurations are described next. The SVM method used was configured with a RBF kernel, similar to previous studies on this dataset [Irrera and Vieira, 2014]. The gamma value for the kernel was set with a relative number: $\frac{1}{n\_features}$. As the computational complexity of SVMs grows at least quadratically with the size of the training data and their dimensionality [Bordes et al., 2005], the data for this algorithm were reduced to a maximum of 40000 samples. The NN used a two-layer architecture with 100 neurons with ReLU as activation function and the Limited-memory Broyden–Fletcher–Goldfarb–Shanno (LBFGS) solver with 0.0001 regularization and a learning rate of 0.001. Finally, the DT used the gini impurity to measure the quality of the split with the CART algorithm. To evaluate the models 10-fold stratified cross-validation was used and each experiment was executed 30 times with different random seeds.

A.3 Individual Failure Modes

The number of methods and configurations considered led to a large number of combinations and thus the discussion focuses on those deemed relevant. Except where stated otherwise, the results refer to the *real* dataset. The values of $\Delta t_l, \Delta t_p$ considered in this analysis are [40, 20], as this was the configuration that systematically achieved the best results. Due to the recurrent use of SVMs in previous OFP studies, their performance will be used as an initial reference for comparison. Similar to existing related work [Irrera and Vieira, 2014], the first experiments attempted to classify *hang* and *crash* samples separately.

When predicting crash failures, SVM models are able to achieve very good results, correctly classifying almost all the non-failure samples and 89% of the *crash* samples. By removing features using correlation, the correct *crash* predictions improved to 92.8%, as illustrated in *Figure A.3.1.* However, when used to predict *hang* failures, they are barely able to distinguish between the samples of the two classes, correctly classifying only 21.2% of the failure samples. In this case, feature selection techniques were not able to significantly improve the results either. Undersampling the data led to a significant improvement, correctly classifying 87.2% of non-failure samples and 80.6% of *hang*. The number of correctly predicted failures can be considered acceptable, although the number of false-positives is significant, reaching ~13% of the total number of non-failure samples. Similar to the observations on the Linux dataset, undersampling the data typically led to models that could more accurately predict failures, at the expense of increasing the number of false-positives.

Other algorithms also performed well for predicting both failure modes, even without pre-processing techniques. When predicting *crash* failures, DT models were able to correctly classify 92.7% of the failure samples. However, the major

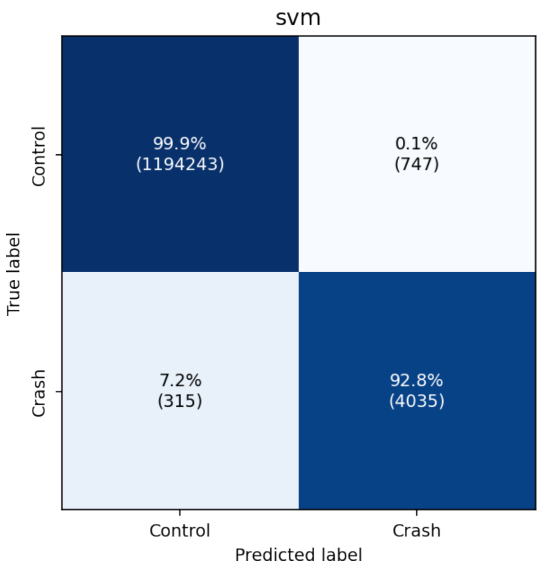


Figure A.3.1: Crash: SVM, Correlation

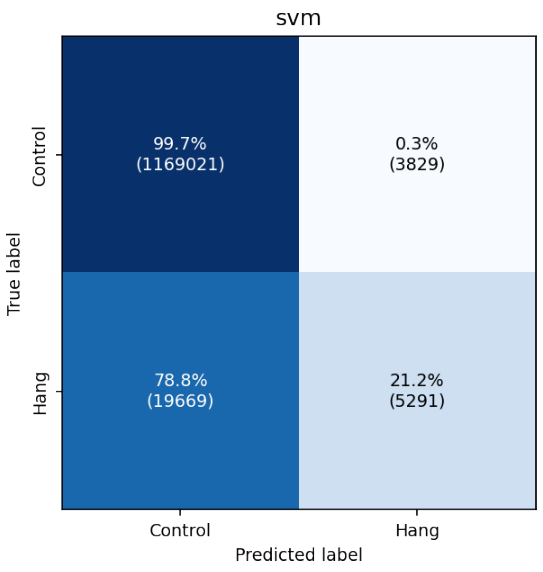


Figure A.3.2: Hang: SVM

difference was with *hang* failures, where it managed to correctly predict 80% of the *hang* samples, while correctly classifying almost all non-failure samples. Using undersampling and feature selection the performance improved, predicting almost all of the *hang* samples, at the expense of a larger number of false-positives, approximately 5%. These observations were similar for the experiments executed with virtualization.

A.4 Multi-Class Classification

When trying to predict the different failures using a single model, SVMs were still unable to perform acceptably, with most of the *crash* and *hang* samples, 74.7%, and 78.8% respectively, being misclassified as non-failure. Using undersampling, the results improved, although not enough, correctly predicting approximately 85.5%, 82.8%, and 87% of *control, hang*, and *crash* samples respectively. DTs were able to separate between the various classes with acceptable performance, correctly classifying almost all the non-failure samples, 80.4% of *hang*, and 92.5% of the *crash* samples. NNs managed to correctly predict almost all the non-failure samples, as well as acceptably distinguish between the two failure modes, correctly predicting approximately 71.4% and 89.1% of the *hang* and *crash* samples respectively. Ensemble methods also achieved good results, with Bagging combined with feature selection by correlation correctly classifying 78.3% of the *hang* and 94.5% of the *crash* samples. Other algorithms, such as Extra Trees and RF, also performed well, albeit not as good as the previously described (e.g., Extra Trees with feature selection by correlation were able to correctly predict almost all non-failure samples, 67.7% of *hang* samples, and 94.9% of *crash* samples).

A.5 Ensembling the Decisions of ML Models

The results obtained in the previous sections were good, but not excellent. Something that was possible to observe for both the Linux and Windows XP datasets

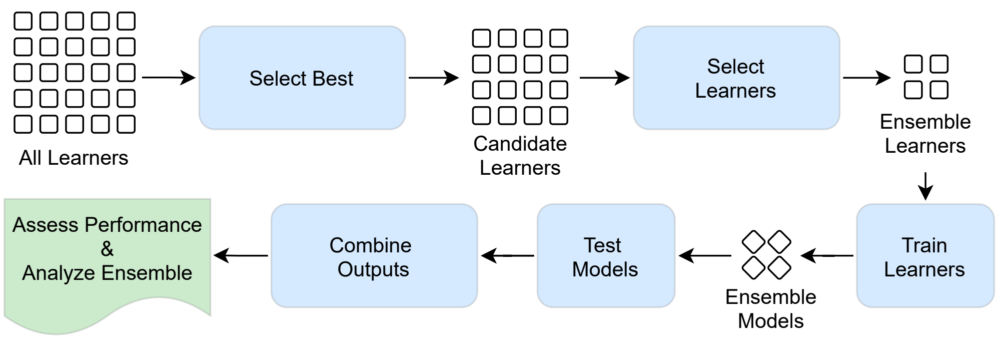


Figure A.5.1: Experimental Process

is that different techniques created models that excel at different parts of the problem. As ensemble methods have shown that they can achieve better results than standalone algorithms, it was decided to explore creating heterogeneous ensembles by combining different ML algorithms and techniques (which may exploit the different biases of each algorithm) [Bian and Wang, 2007; Costa et al., 2018].

A high-level overview of the experimental process is depicted in *Figure A.5.1*. Briefly, after training and assessing the performance of the various classifiers, the best set is chosen for a pool of candidate learners. Then, a selection process chooses the base learners, which are then trained and tested, and their outputs combined using an aggregation method. Finally, the performance of the ensemble is assessed and its structure is analyzed.

A.5.1 Selecting the Base Models

As the purpose of this work is to explore the combination of different learners, various ML algorithms and techniques were considered based on the preliminary results presented in the previous section. The algorithms selected were *DT, Bagging RF, Gradient Boosting (GB)*. To further explore the influence of data sampling several techniques were considered, more precisely, *Random Under/Oversampling, SMOTE*, and *Instance Hardness Threshold*. Although most of the algorithms are already ensembles, in this study their output is used as if they were a single classifier. Their hyperparameters were tuned according to the values described in *Table A.5.1*.

The first stage of this experimental process was to create the individual models, assess their performance, and identify the best solutions. Due to the number of experiments, confusion-matrices are no longer viable. Thus, for this case study the F_2-score metric was chosen.

Compared to the results of the previous section, tuning the hyperparameters of the algorithms allowed most algorithms to improve their 'baseline' (i.e., using the defaults of [Pedregosa et al., 2011]) performance. As an example, tuning Bagging with feature selection by correlation (whose baseline performance, also removing highly correlated features, was 99.8%, 75.6%, and 92% of *control, hang,* and *crash* samples, respectively), it was possible to achieve better results, correctly

Table A.5.1: Algorithms' Hyperparameters

Alg.	**Hyperparameters**
DT	**criter.**: [gini, entropy], **min_samp_split**: [.001, .01, .1, 2], **max_feat.**: [.1, .55, 1.0], **min_samp_leaf**: [.001, .01, .1, 1]
RF	**estimators**: [10, 100, 200], **max_feat.**: [.1, .55, 1.0], **criter.**: [gini, entropy], **min_samp_leaf**: [.001, .01, .1, 1], **min_samp_split**: [.001, .01, .1, 2], **bootstrap**: [1, 0]
Bag.	**max_features**: [.1, .55, 1.0], **bootstrap**: [1, 0], **estimators**: [10, 50, 100, 200]
GB	**estimators**: [50, 100, 200], **learning rate**: [1, .1, .01], **min_samp_leaf**: [.001, .01, .1, 1], **max_feat.**: [.1, .55, 1.0], **min_samp_split**: [.001, .01, .1, 2]

classifying 99.9%, 89%, and 95.4% of *control, hang*, and *crash* samples.

A good ensemble should be composed of diverse models and their combination should reduce incorrect decisions and amplify the correct ones [Polikar, 2006]. To select a diverse set of base learners there are several metrics available (e.g., Q Statistics). However, an issue that arises for this experiment is that the set of algorithms and techniques considered already yield models with very good (and similar) performances, thus, there is not much room for improvement. Additionally, as all the selected models were already very accurate, there was no guarantee that the diversity measures would lead to a better ensemble. This is more blatant when considering the imbalance in the data and the actual number of samples that are correctly predicted.

An obvious approach for choosing the base learners is to combine the best solutions found for each algorithm (i.e., those with the highest performance). The three models selected this way were: *i)* GB with oversampling through SMOTE; *ii)* Bagging also with SMOTE oversampling; and *iii)* RF with Random Oversampling. However, when the results were analyzed (further detailed in *Section A.5.2*) it was possible to conclude that, although they were individually the best, they did not complement each other, resulting in a lower combined performance. A different approach was necessary, which is depicted in *Figure A.5.2*. Additionally, using only the F_2-score to rank the models could make all those selected to be very similar, thus predicting identical parts of the problem. Hence, they were also chosen taking into consideration the Informedness. The top three models of each algorithm for each metric were selected to a pool of potential candidate base learners. This led to a pool of 18 learners. Exploring all the possible combinations of the 18 models for every possible length is infeasible, thus the ensembles were limited to a maximum of **4** learners. This still leads to a significant number of combinations (i.e., 4029 ensembles for each type of output, i.e., crisp labels, and class probabilities).

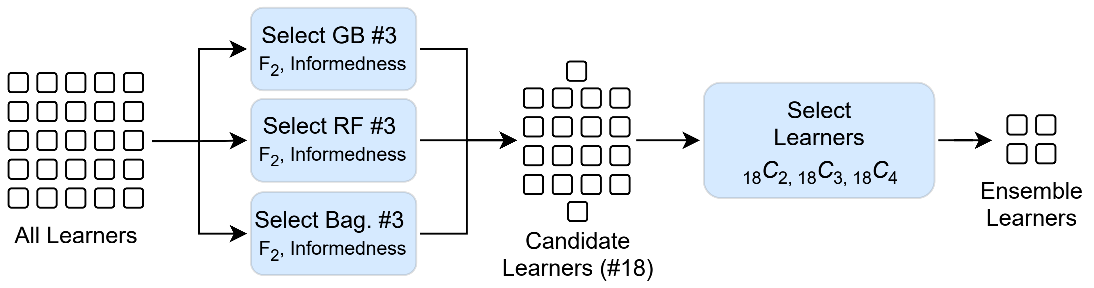


Figure A.5.2: Base Learners Selection

A.5.2 Combining the Base Models

Concerning the combination methods, three approaches were considered: *plurality voting* (i.e., the most voted class is chosen); *soft voting* (i.e., the class prediction probabilities are averaged and the class with the highest probability is chosen); and *stacking* (i.e., a meta-learner is used to model the outputs of the individual learners) for both crisp predictions (i.e., labels) and class probabilities outputs. For the stacking meta-learner, different algorithms were also considered (i.e., DT, RF, Bagging, Logistic Regression, NN). Concerning their configurations, due to time constraints, it was not possible to thoroughly explore and tune all the meta-learner hyperparameters, thus, an ad-hoc approach based on the defaults used by scikit-learn (which in turn are based on literature [Pedregosa et al., 2011]) was used. Notwithstanding, besides the considerable overhead of training and tuning the meta-learner, using stacking did not lead to noticeable improvements and therefore will not be shown here.

Plurality Voting
The initial approach to selecting the learners was to choose the best model for each algorithm (i.e., Bagging, RF, and GB). However, compared to the individual models, the ensemble had a slightly lower performance than the one obtained by GB, correctly predicting 99.8%, 91.4%, and 96% of *control, hang,* and *crash* samples. This was somewhat expected, as the performance of the individual models was already very good and they were created using similar ML techniques.

To explore if using more diverse learners would lead to better performance, the combinations of the 18 candidate base learners were studied. This led to some promising results, as it was possible to observe that there were, in fact, ensembles that outperformed the best individual models. The best ensemble correctly predicted 99.8%, 97.1%, and 98.3% of *control, hang,* and *crash* samples. Although the improvements are not overwhelming this ensemble was able to predict almost 5% more *hang* and 2.1% more *crash* samples. Still, this also came at the expense of slightly more false-positives.

Soft Voting
Although using crisp predictions allowed improving the overall performance (when compared with individual models), it ignores the confidence that the algorithms have in their own predictions. Using the probability outputs allows combining the prediction and the confidence of the base learners [Ting and Witten, 1999].

Once again the performance of the best individual models combined was analyzed. Using the probability outputs it was possible to achieve slightly better performance, correctly classifying 99.9%, 93.6%, and 97.5% of *control, hang*, and *crash* samples, respectively. However, although these results are better than any of the individual models, they are not as good as what was achieved using *plurality voting*. This way, we decided to study the 4029 ensembles of 18 base learners. Results showed that, although some ensembles were able to perform better than the individual models, their performances were not significantly different from the ones observed with *plurality voting* (the best ensemble correctly classified 99.8%, 97.5%, and 98.6% of *control, hang*, and *crash* samples, respectively).

A.5.3 Ensembles Analysis

The previous analyses allowed us to conclude that by combining certain learners it was possible to increase their overall performance. Thus, the next step was focused on analyzing how the ensembles were composed to get some insights into why some models work better as a whole.

Gains Directed Graph
To analyze how the different learners in an ensemble interact, a directed graph between every learner was plotted (similar to *Figure A.5.3*), showing how many different samples were correctly predicted between any two learners.

When analyzing the best ensemble obtained by *soft voting*, it was possible to observe that it was composed of just three models, which are depicted in *Figure A.5.3*. Additionally, (and similar to what was seen for the best ensembles of *plurality voting*) the three models were built using different techniques: one without sampling, another using oversampling, and the third using undersampling. The fact that the ensemble is composed of only three learners also makes it easier to interpret. When comparing the model without sampling with the one that uses oversampling, the latter can predict more failures while correctly predicting almost the same number of non-failure samples. On the other hand, when analyzing the relationships with the model that used undersampling it was possible to observe that it could predict considerably more failures when compared to both the other models. Moreover, it was also possible to conclude that this model not only predicts more failures, it also correctly predicted almost all of the failures that the other two predicted (i.e., only 45 and 17 *hang* different correct predictions were made by the GB and Bagging model respectively). Besides the indication that each of the sampling techniques allows the algorithms to model different parts of the problem, it also suggests that most predictions in the ensemble are supported by at least two algorithms. More precisely, GB with SMOTE and Bagging support and complement the non-failure predictions, while GB with Random undersampling supports and complements the failure predictions of the remaining two learners.

Venn Diagrams
Directed graphs do not allow us to visualize how the different predictions actually overlap between learners. Thus, as some of the best ensembles were composed of only three models, Venn diagrams were plotted for each of the classes to illustrate

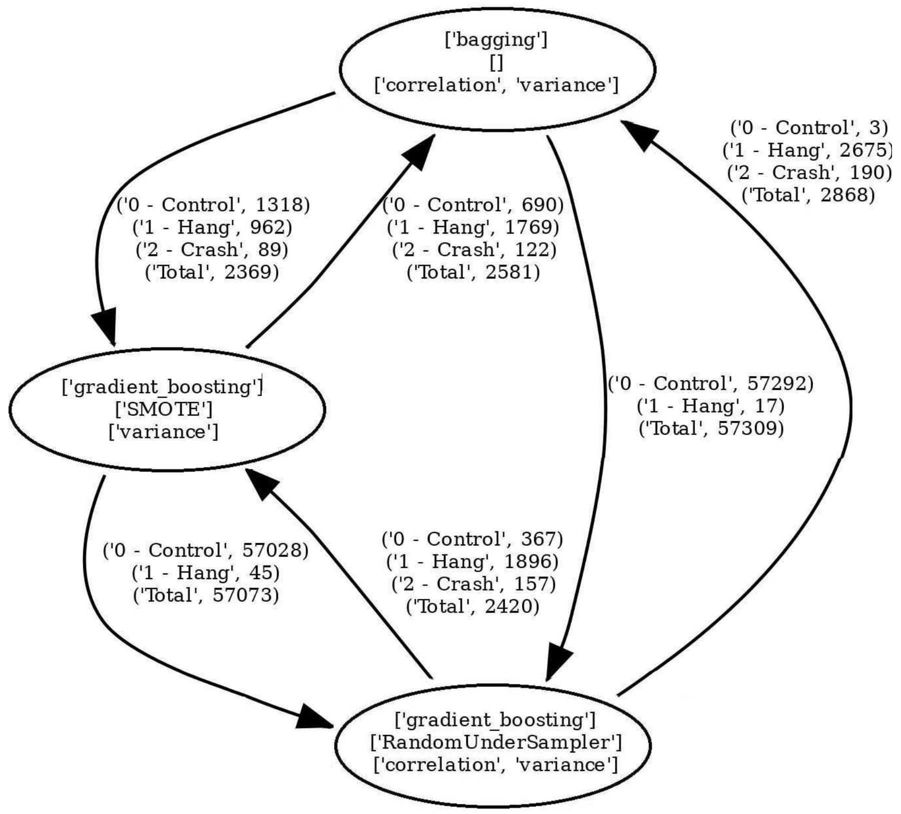


Figure A.5.3: Best Soft Voting Ensemble Graph

how the different predictions relate.

Concerning the ensemble composed of the best individual models (and confirming what was previously assumed), it was possible to observe that although there was a good cover of the problem space by the different learners, there was a considerable number of samples that were only correctly predicted by a single model. On the other hand, when analyzing the diagram for the best *soft voting* ensemble (which can be seen in *Figure A.5.4*; for readability a logarithmic base was used to calculate the areas of the diagrams), it is possible to observe that the learners also complement each other in reaching different samples, but most samples are common to at least two models. In hindsight, this is one of the essential components for a good ensemble, to combine the outputs in a way that correct decisions are amplified [Polikar, 2006]. Hence, although not all models will agree, for most cases at least two will, thus ‘winning’ the vote.

The diagrams for the failure modes illustrated in *Figure A.5.4* show that there is indeed a single model that is able to predict more failures. For the hang failure mode, there are still some samples that are only predicted by the other learners, whilst for the crash failures, such model can predict all the samples the others can and more. However, when analyzing its ability to predict non-failure samples it is possible to observe that the other learners can predict considerably more samples.

To study how the individual predictions relate to the ensemble predictions another set of Venn diagrams was plotted, this time relating the individual (correct)

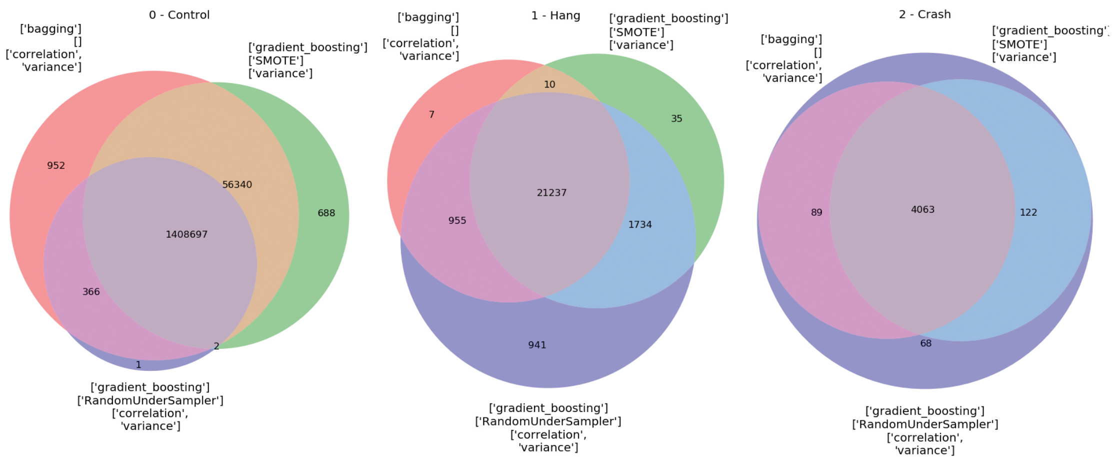


Figure A.5.4: Best Soft Voting Ensemble Venn

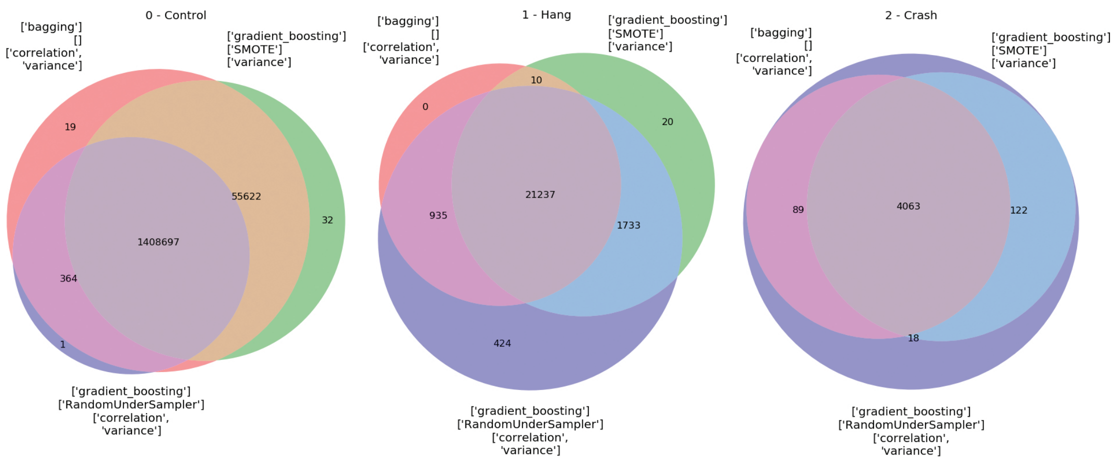


Figure A.5.5: Best Soft Voting Ensemble Prediction Contributions Venn

predictions to those of the ensemble, as shown in *Figure A.5.5*. This validated the hypothesis that decisions made by a single algorithm will less likely be correctly predicted by the ensemble. In fact, most of the predictions that were made only by a single algorithm were not correctly predicted by the ensemble (e.g., concerning the non-failure samples, out of the 952 correct predictions made by Bagging only 19 were also correctly predicted by the ensemble). On the other hand, some of the samples that were correctly predicted would not be possible using crisp outputs (otherwise the two wrong votes would prevail). This suggests that either the correct model was highly confident (enough to trump wrong classifications) or that these are fringe samples, where the learners are not confident of the decision, but their average probabilities lead to a correct prediction.

Diversity Metrics

One of the key components for successful ensembles is that the base learners should be as diverse as possible. The next step was to validate if the best ensembles were, in fact, more diverse than those with a lower performance. The following diversity metrics were computed [Kuncheva and Whitaker, 2003]: *Q Statistics, Correlation Coefficient, Disagreement, Double-fault, Entropy*, and *Interrater Agreement (using Fleiss Kappa [Fleiss, 1981])*. Briefly, the higher the Q Statistics, Correlation

Table A.5.2: Diversity Metrics per Ensemble (Soft Voting)

Ensem. / **Metr.**	Worst Ensem.	Average Ensem.	Best Ensem.
Q Statistics ↓	0.991	0.996	**0.930**
Correlation ↓	0.366	0.538	**0.227**
Disagreement ↑	0.005	0.003	**0.027**
Double Fault ↓	0.001	0.002	**0.001**
Entropy ↑	0.187	0.193	**0.419**
Inter. Agreem. ↓	0.875	0.918	**0.560**

↑ The higher the better (i.e., more diversity)

↓ The lower the better (i.e., more diversity)

Coefficient, Double Fault, and Interrater Agreement, the less diversity there is; and the bigger the Disagreement and Entropy the more diversity exists. All metrics were calculated according to [Kuncheva and Whitaker, 2003] except Entropy, which used a joint entropy formula [Cover and Thomas, 2012].

Three ensembles were analyzed, *i)* an ensemble with lower performance (referred to as *worst* from now on) than that of combining all the best individual models; *ii)* the ensemble composed of the best individual models (referred to as *average* from now on); and *iii)* the best ensemble. The computed metrics can be seen in *Table A.5.2*.

This allowed us to observe that the best ensemble had the highest diversity according to all the metrics. However, comparing the worst and average ensembles, for all metrics besides entropy, the worst ensemble had values that suggest that it is more diverse than the average. Nevertheless, these metrics may be correct, and the worst ensemble may have more diversity. As highlighted by Kuncheva and Whitaker [Kuncheva and Whitaker, 2003], there is often no relationship between diversity and performance. That is, while almost all good ensembles are diverse, diversity per se does not mean that the ensemble will be good.

A.6 Discussion

Similar to the previous study of the Linux OS, it was also possible to create accurate failure predictors for Windows XP. Of all the algorithms considered, tree-based algorithms were once again able to achieve better performances. Overall, from the results of the various algorithms in the different datasets and contexts, predicting hang failures seems more complicated than predicting crash failures on Windows XP. This may be due to the fact that the symptoms in the selected $\Delta t_l, \Delta t_p$ are not as significant as they are for crash failures, thus limiting the distinction between them.

The performance of the different algorithms varied across the different datasets, implying that the behavior of the failures is somewhat different amongst them. Notwithstanding, when gathering all the data in a single dataset the results suggest that it is still possible to accurately distinguish between the different classes. Similar to what was observed with the Linux dataset, this indicates that the behavior of the different failure modes in the various environments is different from each other but nonetheless similar.

The combination of models created using different techniques appears to produce ensembles where they complement each other in a constructive way, thus considerably improving the overall performance with statistical significance (although not shown due to space constraints, the best ensemble was statistically better than both the best individual model and the ensemble composed of the best individual models). *Soft voting* can take advantage of the confidence of the classifiers in their predictions and is, therefore, able to generate better ensembles with fewer base learners. The Venn diagrams for the best ensembles showed that those composed of models that explore different regions but that also overlap with other learners in the ensemble were able to achieve better results (learners that complemented each other but did not overlap were not able to achieve such good performance, possibly because predictions would be canceled out by the remaining learners). Diversity metrics can be used to assess the diversity of the ensembles, but it is not always correlated with performance (hence, the one with the highest diversity will not always be the best). Finally, heterogeneous ensembles explore the different bias of ML algorithms and thus may be a good research direction when individual learners do not reach the intended performance.

Ultimately, this case study provides some external validity to the possibility of using ML algorithms to create accurate failure predictors on complex systems. It also highlights the need to consider a comprehensive set of algorithms and techniques that take the characteristics of the problem into consideration.

Appendix B: Netdata Complete Metrics Set

The complete set of metrics collected by Netdata in every experiment:

```
netdata_apps_cpu_percentage_average
    chart=apps.cpu, family=cpu, dimension=apps.plugin
    chart=apps.cpu, family=cpu, dimension=cron
    chart=apps.cpu, family=cpu, dimension=dhcp
    chart=apps.cpu, family=cpu, dimension=email
    chart=apps.cpu, family=cpu, dimension=go.d.plugin
    chart=apps.cpu, family=cpu, dimension=kernel
    chart=apps.cpu, family=cpu, dimension=ksmd
    chart=apps.cpu, family=cpu, dimension=logs
    chart=apps.cpu, family=cpu, dimension=netdata
    chart=apps.cpu, family=cpu, dimension=other
    chart=apps.cpu, family=cpu, dimension=ssh
    chart=apps.cpu, family=cpu, dimension=system
    chart=apps.cpu, family=cpu, dimension=tc-qos-helper
netdata_apps_cpu_system_percentage_average
    chart=apps.cpu_system, family=cpu, dimension=apps.plugin
    chart=apps.cpu_system, family=cpu, dimension=cron
    chart=apps.cpu_system, family=cpu, dimension=dhcp
    chart=apps.cpu_system, family=cpu, dimension=email
    chart=apps.cpu_system, family=cpu, dimension=go.d.plugin
    chart=apps.cpu_system, family=cpu, dimension=kernel
    chart=apps.cpu_system, family=cpu, dimension=ksmd
    chart=apps.cpu_system, family=cpu, dimension=logs
    chart=apps.cpu_system, family=cpu, dimension=netdata
    chart=apps.cpu_system, family=cpu, dimension=other
    chart=apps.cpu_system, family=cpu, dimension=ssh
    chart=apps.cpu_system, family=cpu, dimension=system
    chart=apps.cpu__system, family=cpu, dimension=tc-qos-helper
netdata__apps__cpu__user__percentage__average
    chart=apps.cpu__user, family=cpu, dimension=apps.plugin
    chart=apps.cpu__user, family=cpu, dimension=cron
    chart=apps.cpu__user, family=cpu, dimension=dhcp
    chart=apps.cpu__user, family=cpu, dimension=email
    chart=apps.cpu__user, family=cpu, dimension=go.d.plugin
    chart=apps.cpu__user, family=cpu, dimension=kernel
    chart=apps.cpu__user, family=cpu, dimension=ksmd
    chart=apps.cpu__user, family=cpu, dimension=logs
    chart=apps.cpu__user, family=cpu, dimension=netdata
    chart=apps.cpu__user, family=cpu, dimension=other
    chart=apps.cpu__user, family=cpu, dimension=ssh
    chart=apps.cpu__user, family=cpu, dimension=system
    chart=apps.cpu__user, family=cpu, dimension=tc-qos-helper
netdata__apps__files__open__files__average
    chart=apps.files, family=disk, dimension=apps.plugin
    chart=apps.files, family=disk, dimension=cron
    chart=apps.files, family=disk, dimension=dhcp
    chart=apps.files, family=disk, dimension=email
    chart=apps.files, family=disk, dimension=go.d.plugin
    chart=apps.files, family=disk, dimension=kernel
    chart=apps.files, family=disk, dimension=ksmd
    chart=apps.files, family=disk, dimension=logs
    chart=apps.files, family=disk, dimension=netdata
    chart=apps.files, family=disk, dimension=other
    chart=apps.files, family=disk, dimension=ssh
    chart=apps.files, family=disk, dimension=system
    chart=apps.files, family=disk, dimension=tc-qos-helper
netdata__apps__lreads__KiB__persec__average
    chart=apps.lreads, family=disk, dimension=apps.plugin
    chart=apps.lreads, family=disk, dimension=cron
    chart=apps.lreads, family=disk, dimension=dhcp
    chart=apps.lreads, family=disk, dimension=email
    chart=apps.lreads, family=disk, dimension=go.d.plugin
    chart=apps.lreads, family=disk, dimension=kernel
    chart=apps.lreads, family=disk, dimension=ksmd
    chart=apps.lreads, family=disk, dimension=logs
    chart=apps.lreads, family=disk, dimension=netdata
    chart=apps.lreads, family=disk, dimension=other
    chart=apps.lreads, family=disk, dimension=ssh
    chart=apps.lreads, family=disk, dimension=system
    chart=apps.lreads, family=disk, dimension=tc-qos-helper
netdata__apps__lwrites__KiB__persec__average
    chart=apps.lwrites, family=disk, dimension=apps.plugin
    chart=apps.lwrites, family=disk, dimension=cron
    chart=apps.lwrites, family=disk, dimension=dhcp
    chart=apps.lwrites, family=disk, dimension=email
    chart=apps.lwrites, family=disk, dimension=go.d.plugin
    chart=apps.lwrites, family=disk, dimension=kernel
    chart=apps.lwrites, family=disk, dimension=ksmd
    chart=apps.lwrites, family=disk, dimension=logs
    chart=apps.lwrites, family=disk, dimension=netdata
    chart=apps.lwrites, family=disk, dimension=other
    chart=apps.lwrites, family=disk, dimension=ssh
    chart=apps.lwrites, family=disk, dimension=system
    chart=apps.lwrites, family=disk, dimension=tc-qos-helper
netdata__apps__major__faults__page__faults__persec__average
    chart=apps.major__faults, family=swap, dimension=apps.plugin
    chart=apps.major__faults, family=swap, dimension=cron
    chart=apps.major__faults, family=swap, dimension=dhcp
    chart=apps.major__faults, family=swap, dimension=email
    chart=apps.major__faults, family=swap, dimension=go.d.plugin
    chart=apps.major__faults, family=swap, dimension=kernel
    chart=apps.major__faults, family=swap, dimension=ksmd
    chart=apps.major__faults, family=swap, dimension=logs
    chart=apps.major__faults, family=swap, dimension=netdata
    chart=apps.major__faults, family=swap, dimension=other
    chart=apps.major__faults, family=swap, dimension=ssh
    chart=apps.major__faults, family=swap, dimension=system
    chart=apps.major__faults, family=swap, dimension=tc-qos-helper
netdata__apps__mem__MiB__average
    chart=apps.mem, family=mem, dimension=apps.plugin
    chart=apps.mem, family=mem, dimension=cron
    chart=apps.mem, family=mem, dimension=dhcp
    chart=apps.mem, family=mem, dimension=email
    chart=apps.mem, family=mem, dimension=go.d.plugin
    chart=apps.mem, family=mem, dimension=kernel
    chart=apps.mem, family=mem, dimension=ksmd
    chart=apps.mem, family=mem, dimension=logs
    chart=apps.mem, family=mem, dimension=netdata
    chart=apps.mem, family=mem, dimension=other
    chart=apps.mem, family=mem, dimension=ssh
    chart=apps.mem, family=mem, dimension=system
    chart=apps.mem, family=mem, dimension=tc-qos-helper
netdata__apps__minor__faults__page__faults__persec__average
    chart=apps.minor__faults, family=mem, dimension=apps.plugin
    chart=apps.minor__faults, family=mem, dimension=cron
    chart=apps.minor__faults, family=mem, dimension=dhcp
    chart=apps.minor__faults, family=mem, dimension=email
    chart=apps.minor__faults, family=mem, dimension=go.d.plugin
    chart=apps.minor__faults, family=mem, dimension=kernel
    chart=apps.minor__faults, family=mem, dimension=ksmd
    chart=apps.minor__faults, family=mem, dimension=logs
    chart=apps.minor_faults, family=mem, dimension=netdata
    chart=apps.minor_faults, family=mem, dimension=other
    chart=apps.minor_faults, family=mem, dimension=ssh
    chart=apps.minor_faults, family=mem, dimension=system
    chart=apps.minor_faults, family=mem, dimension=tc-qos-helper
netdata_apps_pipes_open_pipes_average
    chart=apps.pipes, family=processes, dimension=apps.plugin
    chart=apps.pipes, family=processes, dimension=cron
    chart=apps.pipes, family=processes, dimension=dhcp
    chart=apps.pipes, family=processes, dimension=email
    chart=apps.pipes, family=processes, dimension=go.d.plugin
    chart=apps.pipes, family=processes, dimension=kernel
    chart=apps.pipes, family=processes, dimension=ksmd
    chart=apps.pipes, family=processes, dimension=logs
    chart=apps.pipes, family=processes, dimension=netdata
    chart=apps.pipes, family=processes, dimension=other
    chart=apps.pipes, family=processes, dimension=ssh
    chart=apps.pipes, family=processes, dimension=system
    chart=apps.pipes, family=processes, dimension=tc-qos-helper
netdata_apps_preads_KiB_persec_average
    chart=apps.preads, family=disk, dimension=apps.plugin
    chart=apps.preads, family=disk, dimension=cron
    chart=apps.preads, family=disk, dimension=dhcp
    chart=apps.preads, family=disk, dimension=email
    chart=apps.preads, family=disk, dimension=go.d.plugin
    chart=apps.preads, family=disk, dimension=kernel
    chart=apps.preads, family=disk, dimension=ksmd
    chart=apps.preads, family=disk, dimension=logs
    chart=apps.preads, family=disk, dimension=netdata
    chart=apps.preads, family=disk, dimension=other
    chart=apps.preads, family=disk, dimension=ssh
    chart=apps.preads, family=disk, dimension=system
    chart=apps.preads, family=disk, dimension=tc-qos-helper
netdata_apps_processes_processes_average
    chart=apps.processes, family=processes, dimension=apps.plugin
    chart=apps.processes, family=processes, dimension=cron
    chart=apps.processes, family=processes, dimension=dhcp
    chart=apps.processes, family=processes, dimension=email
    chart=apps.processes, family=processes, dimension=go.d.plugin
    chart=apps.processes, family=processes, dimension=kernel
    chart=apps.processes, family=processes, dimension=ksmd
    chart=apps.processes, family=processes, dimension=logs
    chart=apps.processes, family=processes, dimension=netdata
    chart=apps.processes, family=processes, dimension=other
    chart=apps.processes, family=processes, dimension=ssh
    chart=apps.processes, family=processes, dimension=system
    chart=apps.processes, family=processes, dimension=tc-qos-helper
netdata_apps_pwrites_KiB_persec_average
    chart=apps.pwrites, family=disk, dimension=apps.plugin
    chart=apps.pwrites, family=disk, dimension=cron
    chart=apps.pwrites, family=disk, dimension=dhcp
    chart=apps.pwrites, family=disk, dimension=email
    chart=apps.pwrites, family=disk, dimension=go.d.plugin
    chart=apps.pwrites, family=disk, dimension=kernel
    chart=apps.pwrites, family=disk, dimension=ksmd
    chart=apps.pwrites, family=disk, dimension=logs
    chart=apps.pwrites, family=disk, dimension=netdata
    chart=apps.pwrites, family=disk, dimension=other
    chart=apps.pwrites, family=disk, dimension=ssh
    chart=apps.pwrites, family=disk, dimension=system
    chart=apps.pwrites, family=disk, dimension=tc-qos-helper
netdata_apps_sockets_open_sockets_average
    chart=apps.sockets, family=net, dimension=apps.plugin
    chart=apps.sockets, family=net, dimension=cron
    chart=apps.sockets, family=net, dimension=dhcp
    chart=apps.sockets, family=net, dimension=email
    chart=apps.sockets, family=net, dimension=go.d.plugin
    chart=apps.sockets, family=net, dimension=kernel
    chart=apps.sockets, family=net, dimension=ksmd
    chart=apps.sockets, family=net, dimension=logs
    chart=apps.sockets, family=net, dimension=netdata
    chart=apps.sockets, family=net, dimension=other
    chart=apps.sockets, family=net, dimension=ssh
    chart=apps.sockets, family=net, dimension=system
    chart=apps.sockets, family=net, dimension=tc-qos-helper
netdata_apps_swap_MiB_average
    chart=apps.swap, family=swap, dimension=apps.plugin
    chart=apps.swap, family=swap, dimension=cron
    chart=apps.swap, family=swap, dimension=dhcp
    chart=apps.swap, family=swap, dimension=email
    chart=apps.swap, family=swap, dimension=go.d.plugin
    chart=apps.swap, family=swap, dimension=kernel
    chart=apps.swap, family=swap, dimension=ksmd
    chart=apps.swap, family=swap, dimension=logs
    chart=apps.swap, family=swap, dimension=netdata
    chart=apps.swap, family=swap, dimension=other
    chart=apps.swap, family=swap, dimension=ssh
    chart=apps.swap, family=swap, dimension=system
    chart=apps.swap, family=swap, dimension=tc-qos-helper
netdata_apps_threads_threads_average
    chart=apps.threads, family=processes, dimension=apps.plugin
    chart=apps.threads, family=processes, dimension=cron
    chart=apps.threads, family=processes, dimension=dhcp
    chart=apps.threads, family=processes, dimension=email
    chart=apps.threads, family=processes, dimension=go.d.plugin
    chart=apps.threads, family=processes, dimension=kernel
    chart=apps.threads, family=processes, dimension=ksmd
    chart=apps.threads, family=processes, dimension=logs
    chart=apps.threads, family=processes, dimension=netdata
    chart=apps.threads, family=processes, dimension=other
    chart=apps.threads, family=processes, dimension=ssh
    chart=apps.threads, family=processes, dimension=system
    chart=apps.threads, family=processes, dimension=tc-qos-helper
netdata_apps_uptime_avg_seconds_average
    chart=apps.uptime_avg, family=processes, dimension=apps.plugin
    chart=apps.uptime_avg, family=processes, dimension=cron
    chart=apps.uptime_avg, family=processes, dimension=dhcp
    chart=apps.uptime_avg, family=processes, dimension=email
    chart=apps.uptime_avg, family=processes, dimension=go.d.plugin
    chart=apps.uptime_avg, family=processes, dimension=kernel
    chart=apps.uptime_avg, family=processes, dimension=ksmd
    chart=apps.uptime_avg, family=processes, dimension=logs
    chart=apps.uptime_avg, family=processes, dimension=netdata
    chart=apps.uptime_avg, family=processes, dimension=other
    chart=apps.uptime_avg, family=processes, dimension=ssh
    chart=apps.uptime_avg, family=processes, dimension=system
    chart=apps.uptime_avg, family=processes, dimension=tc-qos-helper
netdata_apps_uptime_max_seconds_average
    chart=apps.uptime_max, family=processes, dimension=apps.plugin
    chart=apps.uptime_max, family=processes, dimension=cron
    chart=apps.uptime_max, family=processes, dimension=dhcp
    chart=apps.uptime_max, family=processes, dimension=email
    chart=apps.uptime_max, family=processes, dimension=go.d.plugin
    chart=apps.uptime_max, family=processes, dimension=kernel
    chart=apps.uptime_max, family=processes, dimension=ksmd
    chart=apps.uptime_max, family=processes, dimension=logs
    chart=apps.uptime_max, family=processes, dimension=netdata
    chart=apps.uptime_max, family=processes, dimension=other
    chart=apps.uptime_max, family=processes, dimension=ssh
    chart=apps.uptime_max, family=processes, dimension=system
    chart=apps.uptime_max, family=processes, dimension=tc-qos-helper
netdata_apps_uptime_min_seconds_average
    chart=apps.uptime_min, family=processes, dimension=apps.plugin
    chart=apps.uptime_min, family=processes, dimension=cron
    chart=apps.uptime_min, family=processes, dimension=dhcp
    chart=apps.uptime_min, family=processes, dimension=email
    chart=apps.uptime_min, family=processes, dimension=go.d.plugin
    chart=apps.uptime_min, family=processes, dimension=kernel
    chart=apps.uptime_min, family=processes, dimension=ksmd
    chart=apps.uptime_min, family=processes, dimension=logs
    chart=apps.uptime_min, family=processes, dimension=netdata
    chart=apps.uptime_min, family=processes, dimension=other
    chart=apps.uptime_min, family=processes, dimension=ssh
    chart=apps.uptime_min, family=processes, dimension=system
    chart=apps.uptime_min, family=processes, dimension=tc-qos-helper
netdata_apps_uptime_seconds_average
    chart=apps.uptime, family=processes, dimension=apps.plugin
    chart=apps.uptime, family=processes, dimension=cron
    chart=apps.uptime, family=processes, dimension=dhcp
    chart=apps.uptime, family=processes, dimension=email
    chart=apps.uptime, family=processes, dimension=go.d.plugin
    chart=apps.uptime, family=processes, dimension=kernel
    chart=apps.uptime, family=processes, dimension=ksmd
    chart=apps.uptime, family=processes, dimension=logs
    chart=apps.uptime, family=processes, dimension=netdata
    chart=apps.uptime, family=processes, dimension=other
    chart=apps.uptime, family=processes, dimension=ssh
    chart=apps.uptime, family=processes, dimension=system
    chart=apps.uptime, family=processes, dimension=tc-qos-helper
netdata_apps_vmem_MiB_average
    chart=apps.vmem, family=mem, dimension=apps.plugin
    chart=apps.vmem, family=mem, dimension=cron
    chart=apps.vmem, family=mem, dimension=dhcp
    chart=apps.vmem, family=mem, dimension=email
    chart=apps.vmem, family=mem, dimension=go.d.plugin
    chart=apps.vmem, family=mem, dimension=kernel
    chart=apps.vmem, family=mem, dimension=ksmd
    chart=apps.vmem, family=mem, dimension=logs
    chart=apps.vmem, family=mem, dimension=netdata
    chart=apps.vmem, family=mem, dimension=other
    chart=apps.vmem, family=mem, dimension=ssh
    chart=apps.vmem, family=mem, dimension=system
    chart=apps.vmem, family=mem, dimension=tc-qos-helper
netdata_cpu_cpu_percentage_average
    chart=cpu.cpu0, family=utilization, dimension=guest
    chart=cpu.cpu0, family=utilization, dimension=guest_nice
    chart=cpu.cpu0, family=utilization, dimension=idle
    chart=cpu.cpu0, family=utilization, dimension=iowait
    chart=cpu.cpu0, family=utilization, dimension=irq
    chart=cpu.cpu0, family=utilization, dimension=nice
    chart=cpu.cpu0, family=utilization, dimension=softirq
    chart=cpu.cpu0, family=utilization, dimension=steal
    chart=cpu.cpu0, family=utilization, dimension=system
    chart=cpu.cpu0, family=utilization, dimension=user
netdata_cpu_interrupts_interrupts_persec_average
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=LOC
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=MCP
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=ata_piix_14
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=ata_piix_15
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=eth0_11
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=floppy_6
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=i8042_1
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=i8042_12
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=rtc0_8
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=serial_4
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=timer_0
netdata_cpu_softirqs_softirqs_persec_average
    chart=cpu.cpu0_softirqs, family=softirqs, dimension=BLOCK
    chart=cpu.cpu0_softirqs, family=softirqs, dimension=HRTIMER
    chart=cpu.cpu0_softirqs, family=softirqs, dimension=NET_RX
    chart=cpu.cpu0_softirqs, family=softirqs, dimension=NET_TX
    chart=cpu.cpu0_softirqs, family=softirqs, dimension=RCU
    chart=cpu.cpu0_softirqs, family=softirqs, dimension=TASKLET
    chart=cpu.cpu0_softirqs, family=softirqs, dimension=TIMER
netdata_cpu_softnet_stat_events_persec_average
    chart=cpu.cpu0_softnet_stat, family=softnet_stat, dimension=dropped
    chart=cpu.cpu0_softnet_stat, family=softnet_stat, dimension=flow_limit_count
    chart=cpu.cpu0_softnet_stat, family=softnet_stat, dimension=processed
    chart=cpu.cpu0_softnet_stat, family=softnet_stat, dimension=received_rps
    chart=cpu.cpu0_softnet_stat, family=softnet_stat, dimension=squeezed
netdata_cpuidle_cpuidle_percentage_average
    chart=cpu.cpu0_cpuidle, family=cpuidle, dimension=C0 (active)
netdata_disk_avgsz_KiB_operation_average
    chart=disk_avgsz.sda, family=sda, dimension=reads
    chart=disk_avgsz.sda, family=sda, dimension=writes
netdata_disk_await_milliseconds_operation_average
    chart=disk_await.sda, family=sda, dimension=reads
    chart=disk_await.sda, family=sda, dimension=writes
netdata_disk_backlog_milliseconds_average
    chart=disk_backlog.sda, family=sda, dimension=backlog
netdata_disk_inodes_inodes_average
    chart=disk_inodes._, family=/, dimension=avail
    chart=disk_inodes._, family=/, dimension=reserved for root
    chart=disk_inodes._, family=/, dimension=used
    chart=disk_inodes._dev, family=/dev, dimension=avail
    chart=disk_inodes._dev, family=/dev, dimension=reserved for root
    chart=disk_inodes._dev, family=/dev, dimension=used
    chart=disk_inodes._run, family=/run, dimension=avail
    chart=disk_inodes._run, family=/run, dimension=reserved for root
    chart=disk_inodes._run, family=/run, dimension=used
    chart=disk_inodes._run_lock, family=/run/lock, dimension=avail
    chart=disk_inodes._run_lock, family=/run/lock, dimension=reserved for root
    chart=disk_inodes._run_lock, family=/run/lock, dimension=used
    chart=disk_inodes._run_shm, family=/run/shm, dimension=avail
    chart=disk_inodes._run_shm, family=/run/shm, dimension=reserved for root
    chart=disk_inodes._run_shm, family=/run/shm, dimension=used
    chart=disk_inodes._run_user, family=/run/user, dimension=avail
    chart=disk_inodes._run_user, family=/run/user, dimension=reserved for root
    chart=disk_inodes._run_user, family=/run/user, dimension=used
netdata_disk_io_KiB_persec_average
    chart=disk.sda, family=sda, dimension=reads
    chart=disk.sda, family=sda, dimension=writes
netdata_disk_iotime_milliseconds_persec_average
    chart=disk_iotime.sda, family=sda, dimension=reads
    chart=disk_iotime.sda, family=sda, dimension=writes
netdata_disk_mops_merged_operations_persec_average
    chart=disk_mops.sda, family=sda, dimension=reads
    chart=disk_mops.sda, family=sda, dimension=writes
netdata_disk_ops_operations_persec_average
    chart=disk_ops.sda, family=sda, dimension=reads
    chart=disk_ops.sda, family=sda, dimension=writes
netdata_disk_qops_operations_average
    chart=disk_qops.sda, family=sda, dimension=operations
netdata_disk_space_GiB_average
    chart=disk_space._, family=/, dimension=avail
    chart=disk_space._, family=/, dimension=reserved for root
    chart=disk_space._, family=/, dimension=used
    chart=disk_space._dev, family=/dev, dimension=avail
    chart=disk_space._dev, family=/dev, dimension=reserved for root
    chart=disk_space._dev, family=/dev, dimension=used
    chart=disk_space._home_fields_fi_local_shared, family=/home/fields/fi_local_shared, dimension=avail
    chart=disk_space._home_fields_fi_local_shared, family=/home/fields/fi_local_shared, dimension=reserved for root
    chart=disk_space._home_fields_fi_local_shared, family=/home/fields/fi_local_shared, dimension=used
    chart=disk_space._run, family=/run, dimension=avail
    chart=disk_space._run, family=/run, dimension=reserved for root
    chart=disk_space._run, family=/run, dimension=used
    chart=disk_space._run_lock, family=/run/lock, dimension=avail
    chart=disk_space._run_lock, family=/run/lock, dimension=reserved for root
    chart=disk_space._run_lock, family=/run/lock, dimension=used
    chart=disk_space._run_shm, family=/run/shm, dimension=avail
    chart=disk_space._run_shm, family=/run/shm, dimension=reserved for root
    chart=disk_space._run_shm, family=/run/shm, dimension=used
    chart=disk_space._run_user, family=/run/user, dimension=avail
    chart=disk_space._run_user, family=/run/user, dimension=reserved for root
    chart=disk_space._run_user, family=/run/user, dimension=used
netdata_disk_svctm_milliseconds_operation_average
    chart=disk_svctm.sda, family=sda, dimension=svctm
netdata_disk_util___of_time_working_average
```

```
    chart=disk_util.sda, family=sda, dimension=utilization
netdata_groups_cpu_percentage_average
    chart=groups.cpu, family=cpu, dimension=daemon
    chart=groups.cpu, family=cpu, dimension=fields
    chart=groups.cpu, family=cpu, dimension=messagebus
    chart=groups.cpu, family=cpu, dimension=netdata
    chart=groups.cpu, family=cpu, dimension=root
    chart=groups.cpu, family=cpu, dimension=smmsp
netdata_groups_cpu_system_percentage_average
    chart=groups.cpu_system, family=cpu, dimension=daemon
    chart=groups.cpu_system, family=cpu, dimension=fields
    chart=groups.cpu_system, family=cpu, dimension=messagebus
    chart=groups.cpu_system, family=cpu, dimension=netdata
    chart=groups.cpu_system, family=cpu, dimension=root
    chart=groups.cpu_system, family=cpu, dimension=smmsp
netdata_groups_cpu_user_percentage_average
    chart=groups.cpu_user, family=cpu, dimension=daemon
    chart=groups.cpu_user, family=cpu, dimension=fields
    chart=groups.cpu_user, family=cpu, dimension=messagebus
    chart=groups.cpu_user, family=cpu, dimension=netdata
    chart=groups.cpu_user, family=cpu, dimension=root
    chart=groups.cpu_user, family=cpu, dimension=smmsp
netdata_groups_files_open_files_average
    chart=groups.files, family=disk, dimension=daemon
    chart=groups.files, family=disk, dimension=fields
    chart=groups.files, family=disk, dimension=messagebus
    chart=groups.files, family=disk, dimension=netdata
    chart=groups.files, family=disk, dimension=root
    chart=groups.files, family=disk, dimension=smmsp
netdata_groups_lreads_KiB_persec_average
    chart=groups.lreads, family=disk, dimension=daemon
    chart=groups.lreads, family=disk, dimension=fields
    chart=groups.lreads, family=disk, dimension=messagebus
    chart=groups.lreads, family=disk, dimension=netdata
    chart=groups.lreads, family=disk, dimension=root
    chart=groups.lreads, family=disk, dimension=smmsp
netdata_groups_lwrites_KiB_persec_average
    chart=groups.lwrites, family=disk, dimension=daemon
    chart=groups.lwrites, family=disk, dimension=fields
    chart=groups.lwrites, family=disk, dimension=messagebus
    chart=groups.lwrites, family=disk, dimension=netdata
    chart=groups.lwrites, family=disk, dimension=root
    chart=groups.lwrites, family=disk, dimension=smmsp
netdata_groups_major_faults_page_faults_persec_average
    chart=groups.major_faults, family=swap, dimension=daemon
    chart=groups.major_faults, family=swap, dimension=fields
    chart=groups.major_faults, family=swap, dimension=messagebus
    chart=groups.major_faults, family=swap, dimension=netdata
    chart=groups.major_faults, family=swap, dimension=root
    chart=groups.major_faults, family=swap, dimension=smmsp
netdata_groups_mem_MiB_average
    chart=groups.mem, family=mem, dimension=daemon
    chart=groups.mem, family=mem, dimension=fields
    chart=groups.mem, family=mem, dimension=messagebus
    chart=groups.mem, family=mem, dimension=netdata
    chart=groups.mem, family=mem, dimension=root
    chart=groups.mem, family=mem, dimension=smmsp
netdata_groups_minor_faults_page_faults_persec_average
    chart=groups.minor_faults, family=mem, dimension=daemon
    chart=groups.minor_faults, family=mem, dimension=fields
    chart=groups.minor_faults, family=mem, dimension=messagebus
    chart=groups.minor_faults, family=mem, dimension=netdata
    chart=groups.minor_faults, family=mem, dimension=root
    chart=groups.minor_faults, family=mem, dimension=smmsp
netdata_groups_pipes_open_pipes_average
    chart=groups.pipes, family=processes, dimension=daemon
    chart=groups.pipes, family=processes, dimension=fields
    chart=groups.pipes, family=processes, dimension=messagebus
    chart=groups.pipes, family=processes, dimension=netdata
    chart=groups.pipes, family=processes, dimension=root
    chart=groups.pipes, family=processes, dimension=smmsp
netdata_groups_preads_KiB_persec_average
    chart=groups.preads, family=disk, dimension=daemon
    chart=groups.preads, family=disk, dimension=fields
    chart=groups.preads, family=disk, dimension=messagebus
    chart=groups.preads, family=disk, dimension=netdata
    chart=groups.preads, family=disk, dimension=root
    chart=groups.preads, family=disk, dimension=smmsp
netdata_groups_processes_processes_average
    chart=groups.processes, family=processes, dimension=daemon
    chart=groups.processes, family=processes, dimension=fields
    chart=groups.processes, family=processes, dimension=messagebus
    chart=groups.processes, family=processes, dimension=netdata
    chart=groups.processes, family=processes, dimension=root
    chart=groups.processes, family=processes, dimension=smmsp
netdata_groups_pwrites_KiB_persec_average
    chart=groups.pwrites, family=disk, dimension=daemon
    chart=groups.pwrites, family=disk, dimension=fields
    chart=groups.pwrites, family=disk, dimension=messagebus
    chart=groups.pwrites, family=disk, dimension=netdata
    chart=groups.pwrites, family=disk, dimension=root
    chart=groups.pwrites, family=disk, dimension=smmsp
netdata_groups_sockets_open_sockets_average
    chart=groups.sockets, family=net, dimension=daemon
    chart=groups.sockets, family=net, dimension=fields
    chart=groups.sockets, family=net, dimension=messagebus
    chart=groups.sockets, family=net, dimension=netdata
    chart=groups.sockets, family=net, dimension=root
    chart=groups.sockets, family=net, dimension=smmsp
netdata_groups_swap_MiB_average
    chart=groups.swap, family=swap, dimension=daemon
    chart=groups.swap, family=swap, dimension=fields
    chart=groups.swap, family=swap, dimension=messagebus
    chart=groups.swap, family=swap, dimension=netdata
    chart=groups.swap, family=swap, dimension=root
    chart=groups.swap, family=swap, dimension=smmsp
netdata_groups_threads_threads_average
    chart=groups.threads, family=processes, dimension=daemon
    chart=groups.threads, family=processes, dimension=fields
    chart=groups.threads, family=processes, dimension=messagebus
    chart=groups.threads, family=processes, dimension=netdata
    chart=groups.threads, family=processes, dimension=root
    chart=groups.threads, family=processes, dimension=smmsp
netdata_groups_uptime_avg_seconds_average
    chart=groups.uptime_avg, family=processes, dimension=daemon
    chart=groups.uptime_avg, family=processes, dimension=fields
    chart=groups.uptime_avg, family=processes, dimension=messagebus
    chart=groups.uptime_avg, family=processes, dimension=netdata
    chart=groups.uptime_avg, family=processes, dimension=root
    chart=groups.uptime_avg, family=processes, dimension=smmsp
netdata_groups_uptime_max_seconds_average
    chart=groups.uptime_max, family=processes, dimension=daemon
    chart=groups.uptime_max, family=processes, dimension=fields
    chart=groups.uptime_max, family=processes, dimension=messagebus
    chart=groups.uptime_max, family=processes, dimension=netdata
    chart=groups.uptime_max, family=processes, dimension=root
    chart=groups.uptime_max, family=processes, dimension=smmsp
netdata_groups_uptime_min_seconds_average
    chart=groups.uptime_min, family=processes, dimension=daemon
    chart=groups.uptime_min, family=processes, dimension=fields
    chart=groups.uptime_min, family=processes, dimension=messagebus
    chart=groups.uptime_min, family=processes, dimension=netdata
    chart=groups.uptime_min, family=processes, dimension=root
    chart=groups.uptime_min, family=processes, dimension=smmsp
netdata_groups_uptime_seconds_average
    chart=groups.uptime, family=processes, dimension=daemon
    chart=groups.uptime, family=processes, dimension=fields
    chart=groups.uptime, family=processes, dimension=messagebus
    chart=groups.uptime, family=processes, dimension=netdata
    chart=groups.uptime, family=processes, dimension=root
    chart=groups.uptime, family=processes, dimension=smmsp
netdata_groups_vmem_MiB_average
    chart=groups.vmem, family=mem, dimension=daemon
    chart=groups.vmem, family=mem, dimension=fields
    chart=groups.vmem, family=mem, dimension=messagebus
    chart=groups.vmem, family=mem, dimension=netdata
    chart=groups.vmem, family=mem, dimension=root
    chart=groups.vmem, family=mem, dimension=smmsp
netdata_info
    instance=ubuntu, application=netdata, version=v1.20.0-167-nightly
netdata_ip_ecnpkts_packets_persec_average
    chart=ip.ecnpkts, family=ecn, dimension=CEP
    chart=ip.ecnpkts, family=ecn, dimension=ECTP0
    chart=ip.ecnpkts, family=ecn, dimension=ECTP1
    chart=ip.ecnpkts, family=ecn, dimension=NoECTP
netdata_ip_tcpofo_packets_persec_average
    chart=ip.tcpofo, family=tcp, dimension=dropped
    chart=ip.tcpofo, family=tcp, dimension=inqueue
    chart=ip.tcpofo, family=tcp, dimension=merged
    chart=ip.tcpofo, family=tcp, dimension=pruned
netdata_ipv4_icmp_errors_packets_persec_average
    chart=ipv4.icmp_errors, family=icmp, dimension=InCsumErrors
    chart=ipv4.icmp_errors, family=icmp, dimension=InErrors
    chart=ipv4.icmp_errors, family=icmp, dimension=OutErrors
netdata_ipv4_icmp_packets_persec_average
    chart=ipv4.icmp, family=icmp, dimension=received
    chart=ipv4.icmp, family=icmp, dimension=sent
netdata_ipv4_icmpmsg_packets_persec_average
    chart=ipv4.icmpmsg, family=icmp, dimension=InDestUnreachs
    chart=ipv4.icmpmsg, family=icmp, dimension=InEchoReps
    chart=ipv4.icmpmsg, family=icmp, dimension=InEchos
    chart=ipv4.icmpmsg, family=icmp, dimension=InParmProbs
    chart=ipv4.icmpmsg, family=icmp, dimension=InRedirects
    chart=ipv4.icmpmsg, family=icmp, dimension=InRouterAdvert
    chart=ipv4.icmpmsg, family=icmp, dimension=InRouterSelect
    chart=ipv4.icmpmsg, family=icmp, dimension=InTimeExcds
    chart=ipv4.icmpmsg, family=icmp, dimension=InTimestampReps
    chart=ipv4.icmpmsg, family=icmp, dimension=InTimestamps
    chart=ipv4.icmpmsg, family=icmp, dimension=OutDestUnreachs
    chart=ipv4.icmpmsg, family=icmp, dimension=OutEchoReps
    chart=ipv4.icmpmsg, family=icmp, dimension=OutEchos
    chart=ipv4.icmpmsg, family=icmp, dimension=OutParmProbs
    chart=ipv4.icmpmsg, family=icmp, dimension=OutRedirects
    chart=ipv4.icmpmsg, family=icmp, dimension=OutRouterAdvert
    chart=ipv4.icmpmsg, family=icmp, dimension=OutRouterSelect
    chart=ipv4.icmpmsg, family=icmp, dimension=OutTimeExcds
    chart=ipv4.icmpmsg, family=icmp, dimension=OutTimestampReps
    chart=ipv4.icmpmsg, family=icmp, dimension=OutTimestamps
netdata_ipv4_packets_packets_persec_average
    chart=ipv4.packets, family=packets, dimension=delivered
    chart=ipv4.packets, family=packets, dimension=forwarded
    chart=ipv4.packets, family=packets, dimension=received
    chart=ipv4.packets, family=packets, dimension=sent
netdata_ipv4_sockstat_sockets_sockets_average
    chart=ipv4.sockstat_sockets, family=sockets, dimension=used
netdata_ipv4_sockstat_tcp_mem_KiB_average
    chart=ipv4.sockstat_tcp_mem, family=tcp, dimension=mem
netdata_ipv4_sockstat_tcp_sockets_sockets_average
    chart=ipv4.sockstat_tcp_sockets, family=tcp, dimension=alloc
    chart=ipv4.sockstat_tcp_sockets, family=tcp, dimension=inuse
    chart=ipv4.sockstat_tcp_sockets, family=tcp, dimension=orphan
    chart=ipv4.sockstat_tcp_sockets, family=tcp, dimension=timewait
netdata_ipv4_sockstat_udp_sockets_sockets_average
    chart=ipv4.sockstat_udp_sockets, family=udp, dimension=inuse
netdata_ipv4_tcphandshake_events_persec_average
    chart=ipv4.tcphandshake, family=tcp, dimension=AttemptFails
    chart=ipv4.tcphandshake, family=tcp, dimension=EstabResets
    chart=ipv4.tcphandshake, family=tcp, dimension=OutRsts
    chart=ipv4.tcphandshake, family=tcp, dimension=SynRetrans
netdata_ipv4_tcpopens_connections_persec_average
    chart=ipv4.tcpopens, family=tcp, dimension=active
    chart=ipv4.tcpopens, family=tcp, dimension=passive
netdata_ipv4_tcppackets_packets_persec_average
    chart=ipv4.tcppackets, family=tcp, dimension=received
    chart=ipv4.tcppackets, family=tcp, dimension=sent
netdata_ipv4_tcpsock_active_connections_average
    chart=ipv4.tcpsock, family=tcp, dimension=connections
netdata_ipv4_udperrors_events_persec_average
    chart=ipv4.udperrors, family=udp, dimension=IgnoredMulti
    chart=ipv4.udperrors, family=udp, dimension=InCsumErrors
    chart=ipv4.udperrors, family=udp, dimension=InErrors
    chart=ipv4.udperrors, family=udp, dimension=NoPorts
    chart=ipv4.udperrors, family=udp, dimension=RcvbufErrors
    chart=ipv4.udperrors, family=udp, dimension=SndbufErrors
netdata_ipv4_udppackets_packets_persec_average
    chart=ipv4.udppackets, family=udp, dimension=received
    chart=ipv4.udppackets, family=udp, dimension=sent
netdata_ipv6_ect_packets_persec_average
    chart=ipv6.ect, family=packets, dimension=InCEPkts
    chart=ipv6.ect, family=packets, dimension=InECT0Pkts
    chart=ipv6.ect, family=packets, dimension=InECT1Pkts
    chart=ipv6.ect, family=packets, dimension=InNoECTPkts
netdata_ipv6_icmp_messages_persec_average
    chart=ipv6.icmp, family=icmp6, dimension=received
    chart=ipv6.icmp, family=icmp6, dimension=sent
netdata_ipv6_icmperrors_errors_persec_average
    chart=ipv6.icmperrors, family=icmp6, dimension=InCsumErrors
    chart=ipv6.icmperrors, family=icmp6, dimension=InDestUnreachs
    chart=ipv6.icmperrors, family=icmp6, dimension=InErrors
    chart=ipv6.icmperrors, family=icmp6, dimension=InParmProblems
    chart=ipv6.icmperrors, family=icmp6, dimension=InPktTooBigs
    chart=ipv6.icmperrors, family=icmp6, dimension=InTimeExcds
    chart=ipv6.icmperrors, family=icmp6, dimension=OutDestUnreachs
    chart=ipv6.icmperrors, family=icmp6, dimension=OutErrors
    chart=ipv6.icmperrors, family=icmp6, dimension=OutParmProblems
    chart=ipv6.icmperrors, family=icmp6, dimension=OutPktTooBigs
    chart=ipv6.icmperrors, family=icmp6, dimension=OutTimeExcds
netdata_ipv6_icmpmldv2_reports_persec_average
    chart=ipv6.icmpmldv2, family=icmp6, dimension=received
    chart=ipv6.icmpmldv2, family=icmp6, dimension=sent
netdata_ipv6_icmpneighbor_messages_persec_average
    chart=ipv6.icmpneighbor, family=icmp6, dimension=InAdvertisements
    chart=ipv6.icmpneighbor, family=icmp6, dimension=InSolicits
    chart=ipv6.icmpneighbor, family=icmp6, dimension=OutAdvertisements
    chart=ipv6.icmpneighbor, family=icmp6, dimension=OutSolicits
netdata_ipv6_icmprouter_messages_persec_average
    chart=ipv6.icmprouter, family=icmp6, dimension=InAdvertisements
    chart=ipv6.icmprouter, family=icmp6, dimension=InSolicits
    chart=ipv6.icmprouter, family=icmp6, dimension=OutAdvertisements
    chart=ipv6.icmprouter, family=icmp6, dimension=OutSolicits
netdata_ipv6_icmptypes_messages_persec_average
    chart=ipv6.icmptypes, family=icmp6, dimension=InType1
    chart=ipv6.icmptypes, family=icmp6, dimension=InType128
    chart=ipv6.icmptypes, family=icmp6, dimension=InType129
    chart=ipv6.icmptypes, family=icmp6, dimension=InType136
    chart=ipv6.icmptypes, family=icmp6, dimension=OutType1
    chart=ipv6.icmptypes, family=icmp6, dimension=OutType128
    chart=ipv6.icmptypes, family=icmp6, dimension=OutType129
    chart=ipv6.icmptypes, family=icmp6, dimension=OutType133
    chart=ipv6.icmptypes, family=icmp6, dimension=OutType135
    chart=ipv6.icmptypes, family=icmp6, dimension=OutType143
netdata_ipv6_mcast_kilobits_persec_average
    chart=ipv6.mcast, family=multicast6, dimension=received
    chart=ipv6.mcast, family=multicast6, dimension=sent
netdata_ipv6_mcastpkts_packets_persec_average
    chart=ipv6.mcastpkts, family=multicast6, dimension=received
    chart=ipv6.mcastpkts, family=multicast6, dimension=sent
netdata_ipv6_packets_packets_persec_average
    chart=ipv6.packets, family=packets, dimension=delivers
    chart=ipv6.packets, family=packets, dimension=forwarded
    chart=ipv6.packets, family=packets, dimension=received
    chart=ipv6.packets, family=packets, dimension=sent
netdata_ipv6_sockstat6_tcp_sockets_sockets_average
    chart=ipv6.sockstat6_tcp_sockets, family=tcp6, dimension=inuse
netdata_ipv6_sockstat6_udp_sockets_sockets_average
    chart=ipv6.sockstat6_udp_sockets, family=udp6, dimension=inuse
netdata_ipv6_udperrors_events_persec_average
    chart=ipv6.udperrors, family=udp6, dimension=IgnoredMulti
    chart=ipv6.udperrors, family=udp6, dimension=InCsumErrors
    chart=ipv6.udperrors, family=udp6, dimension=InErrors
    chart=ipv6.udperrors, family=udp6, dimension=NoPorts
    chart=ipv6.udperrors, family=udp6, dimension=RcvbufErrors
    chart=ipv6.udperrors, family=udp6, dimension=SndbufErrors
netdata_ipv6_udppackets_packets_persec_average
    chart=ipv6.udppackets, family=udp6, dimension=received
    chart=ipv6.udppackets, family=udp6, dimension=sent
netdata_mem_available_MiB_average
    chart=mem.available, family=system, dimension=avail
netdata_mem_committed_MiB_average
    chart=mem.committed, family=system, dimension=Committed_AS
netdata_mem_kernel_MiB_average
    chart=mem.kernel, family=kernel, dimension=KernelStack
    chart=mem.kernel, family=kernel, dimension=PageTables
    chart=mem.kernel, family=kernel, dimension=Slab
    chart=mem.kernel, family=kernel, dimension=VmallocUsed
netdata_mem_pgfaults_faults_persec_average
    chart=mem.pgfaults, family=system, dimension=major
    chart=mem.pgfaults, family=system, dimension=minor
netdata_mem_slab_MiB_average
    chart=mem.slab, family=slab, dimension=reclaimable
    chart=mem.slab, family=slab, dimension=unreclaimable
netdata_mem_writeback_MiB_average
    chart=mem.writeback, family=kernel, dimension=Bounce
    chart=mem.writeback, family=kernel, dimension=Dirty
    chart=mem.writeback, family=kernel, dimension=FuseWriteback
    chart=mem.writeback, family=kernel, dimension=NfsWriteback
    chart=mem.writeback, family=kernel, dimension=Writeback
netdata_net_errors_errors_persec_average
    chart=net_errors.eth0, family=eth0, dimension=inbound
    chart=net_errors.eth0, family=eth0, dimension=outbound
netdata_net_events_events_persec_average
    chart=net_events.eth0, family=eth0, dimension=carrier
    chart=net_events.eth0, family=eth0, dimension=collisions
    chart=net_events.eth0, family=eth0, dimension=frames
netdata_net_net_kilobits_persec_average
    chart=net.eth0, family=eth0, dimension=received
    chart=net.eth0, family=eth0, dimension=sent
netdata_net_packets_packets_persec_average
    chart=net_packets.eth0, family=eth0, dimension=multicast
    chart=net_packets.eth0, family=eth0, dimension=received
    chart=net_packets.eth0, family=eth0, dimension=sent
netdata_netdata_apps_children_fix_percentage_average
    chart=netdata.apps_children_fix, family=apps.plugin, dimension=cgtime
    chart=netdata.apps_children_fix, family=apps.plugin, dimension=cmajflt
    chart=netdata.apps_children_fix, family=apps.plugin, dimension=cminflt
    chart=netdata.apps_children_fix, family=apps.plugin, dimension=cstime
    chart=netdata.apps_children_fix, family=apps.plugin, dimension=cutime
netdata_netdata_apps_cpu_milliseconds_persec_average
    chart=netdata.apps_cpu, family=apps.plugin, dimension=system
    chart=netdata.apps_cpu, family=apps.plugin, dimension=user
netdata_netdata_apps_fix_percentage_average
    chart=netdata.apps_fix, family=apps.plugin, dimension=gtime
    chart=netdata.apps_fix, family=apps.plugin, dimension=majflt
    chart=netdata.apps_fix, family=apps.plugin, dimension=minflt
    chart=netdata.apps_fix, family=apps.plugin, dimension=stime
    chart=netdata.apps_fix, family=apps.plugin, dimension=utime
netdata_netdata_apps_sizes_files_persec_average
    chart=netdata.apps_sizes, family=apps.plugin, dimension=calls
    chart=netdata.apps_sizes, family=apps.plugin, dimension=fds
    chart=netdata.apps_sizes, family=apps.plugin, dimension=filenames
    chart=netdata.apps_sizes, family=apps.plugin, dimension=files
    chart=netdata.apps_sizes, family=apps.plugin, dimension=inode_changes
    chart=netdata.apps_sizes, family=apps.plugin, dimension=link_changes
    chart=netdata.apps_sizes, family=apps.plugin, dimension=new pids
    chart=netdata.apps_sizes, family=apps.plugin, dimension=pids
    chart=netdata.apps_sizes, family=apps.plugin, dimension=targets
```

```
netdata_netdata_clients_connected_clients_average
    chart=netdata.clients, family=netdata, dimension=clients
netdata_netdata_compression_ratio_percentage_average
    chart=netdata.compression_ratio, family=netdata, dimension=savings
netdata_netdata_db_points_points_persec_average
    chart=netdata.db_points, family=queries, dimension=generated
    chart=netdata.db_points, family=queries, dimension=read
netdata_netdata_net_kilobits_persec_average
    chart=netdata.net, family=netdata, dimension=in
    chart=netdata.net, family=netdata, dimension=out
netdata_netdata_plugin_cgroups_cpu_milliseconds_persec_average
    chart=netdata.plugin_cgroups_cpu, family=cgroups, dimension=system
    chart=netdata.plugin_cgroups_cpu, family=cgroups, dimension=user
netdata_netdata_plugin_diskspace_dt_milliseconds_run_average
    chart=netdata.plugin_diskspace_dt, family=diskspace, dimension=duration
netdata_netdata_plugin_diskspace_milliseconds_persec_average
    chart=netdata.plugin_diskspace, family=diskspace, dimension=system
    chart=netdata.plugin_diskspace, family=diskspace, dimension=user
netdata_netdata_plugin_proc_cpu_milliseconds_persec_average
    chart=netdata.plugin_proc_cpu, family=proc, dimension=system
    chart=netdata.plugin_proc_cpu, family=proc, dimension=user
netdata_netdata_plugin_proc_modules_milliseconds_run_average
    chart=netdata.plugin_proc_modules, family=proc, dimension=btrfs
    chart=netdata.plugin_proc_modules, family=proc, dimension=diskstats
    chart=netdata.plugin_proc_modules, family=proc, dimension=entropy
    chart=netdata.plugin_proc_modules, family=proc, dimension=interrupts
    chart=netdata.plugin_proc_modules, family=proc, dimension=ipc
    chart=netdata.plugin_proc_modules, family=proc, dimension=ksm
    chart=netdata.plugin_proc_modules, family=proc, dimension=loadavg
    chart=netdata.plugin_proc_modules, family=proc, dimension=mdstat
    chart=netdata.plugin_proc_modules, family=proc, dimension=meminfo
    chart=netdata.plugin_proc_modules, family=proc, dimension=netdev
    chart=netdata.plugin_proc_modules, family=proc, dimension=netstat
    chart=netdata.plugin_proc_modules, family=proc, dimension=power_supply
    chart=netdata.plugin_proc_modules, family=proc, dimension=snmp
    chart=netdata.plugin_proc_modules, family=proc, dimension=snmp6
    chart=netdata.plugin_proc_modules, family=proc, dimension=sockstat
    chart=netdata.plugin_proc_modules, family=proc, dimension=sockstat6
    chart=netdata.plugin_proc_modules, family=proc, dimension=softirqs
    chart=netdata.plugin_proc_modules, family=proc, dimension=softnet
    chart=netdata.plugin_proc_modules, family=proc, dimension=stat
    chart=netdata.plugin_proc_modules, family=proc, dimension=uptime
    chart=netdata.plugin_proc_modules, family=proc, dimension=vmstat
netdata_netdata_plugin_tc_cpu_milliseconds_persec_average
    chart=netdata.plugin_tc_cpu, family=tc.helper, dimension=system
    chart=netdata.plugin_tc_cpu, family=tc.helper, dimension=user
netdata_netdata_plugin_tc_time_milliseconds_run_average
    chart=netdata.plugin_tc_time, family=tc.helper, dimension=run time
netdata_netdata_private_charts_charts_average
    chart=netdata.private_charts, family=statsd, dimension=charts
netdata_netdata_queries_queries_persec_average
    chart=netdata.queries, family=queries, dimension=queries
netdata_netdata_requests_requests_persec_average
    chart=netdata.requests, family=netdata, dimension=requests
netdata_netdata_response_time_milliseconds_request_average
    chart=netdata.response_time, family=netdata, dimension=average
    chart=netdata.response_time, family=netdata, dimension=max
netdata_netdata_server_cpu_milliseconds_persec_average
    chart=netdata.server_cpu, family=netdata, dimension=system
    chart=netdata.server_cpu, family=netdata, dimension=user
netdata_netdata_statsd_bytes_kilobits_persec_average
    chart=netdata.statsd_bytes, family=statsd, dimension=tcp
    chart=netdata.statsd_bytes, family=statsd, dimension=udp
netdata_netdata_statsd_cpu_milliseconds_persec_average
    chart=netdata.plugin_statsd_charting_cpu, family=statsd, dimension=system
    chart=netdata.plugin_statsd_charting_cpu, family=statsd, dimension=user
    chart=netdata.plugin_statsd_collector1_cpu, family=statsd, dimension=system
    chart=netdata.plugin_statsd_collector1_cpu, family=statsd, dimension=user
netdata_netdata_statsd_events_events_persec_average
    chart=netdata.statsd_events, family=statsd, dimension=counters
    chart=netdata.statsd_events, family=statsd, dimension=errors
    chart=netdata.statsd_events, family=statsd, dimension=gauges
    chart=netdata.statsd_events, family=statsd, dimension=histograms
    chart=netdata.statsd_events, family=statsd, dimension=meters
    chart=netdata.statsd_events, family=statsd, dimension=sets
    chart=netdata.statsd_events, family=statsd, dimension=timers
    chart=netdata.statsd_events, family=statsd, dimension=unknown
netdata_netdata_statsd_metrics_metrics_average
    chart=netdata.statsd_metrics, family=statsd, dimension=counters
    chart=netdata.statsd_metrics, family=statsd, dimension=gauges
    chart=netdata.statsd_metrics, family=statsd, dimension=histograms
    chart=netdata.statsd_metrics, family=statsd, dimension=meters
    chart=netdata.statsd_metrics, family=statsd, dimension=sets
    chart=netdata.statsd_metrics, family=statsd, dimension=timers
netdata_netdata_statsd_packets_packets_persec_average
    chart=netdata.statsd_packets, family=statsd, dimension=tcp
    chart=netdata.statsd_packets, family=statsd, dimension=udp
```

```
netdata_netdata_statsd_reads_reads_persec_average
    chart=netdata.statsd_reads, family=statsd, dimension=tcp
    chart=netdata.statsd_reads, family=statsd, dimension=udp
netdata_netdata_statsd_useful_metrics_metrics_average
    chart=netdata.statsd_useful_metrics, family=statsd, dimension=counters
    chart=netdata.statsd_useful_metrics, family=statsd, dimension=gauges
    chart=netdata.statsd_useful_metrics, family=statsd, dimension=histograms
    chart=netdata.statsd_useful_metrics, family=statsd, dimension=meters
    chart=netdata.statsd_useful_metrics, family=statsd, dimension=sets
    chart=netdata.statsd_useful_metrics, family=statsd, dimension=timers
netdata_netdata_tcp_connected_sockets_average
    chart=netdata.tcp_connected, family=statsd, dimension=connected
netdata_netdata_tcp_connects_events_average
    chart=netdata.tcp_connects, family=statsd, dimension=connects
    chart=netdata.tcp_connects, family=statsd, dimension=disconnects
netdata_netdata_web_cpu_milliseconds_persec_average
    chart=netdata.web_thread1_cpu, family=web, dimension=system
    chart=netdata.web_thread1_cpu, family=web, dimension=user
netdata_system_active_processes_processes_average
    chart=system.active_processes, family=processes, dimension=active
netdata_system_cpu_percentage_average
    chart=system.cpu, family=cpu, dimension=guest
    chart=system.cpu, family=cpu, dimension=guest_nice
    chart=system.cpu, family=cpu, dimension=idle
    chart=system.cpu, family=cpu, dimension=iowait
    chart=system.cpu, family=cpu, dimension=irq
    chart=system.cpu, family=cpu, dimension=nice
    chart=system.cpu, family=cpu, dimension=softirq
    chart=system.cpu, family=cpu, dimension=steal
    chart=system.cpu, family=cpu, dimension=system
    chart=system.cpu, family=cpu, dimension=user
netdata_system_ctxt_context_switches_persec_average
    chart=system.ctxt, family=processes, dimension=switches
netdata_system_entropy_entropy_average
    chart=system.entropy, family=entropy, dimension=entropy
netdata_system_forks_processes_persec_average
    chart=system.forks, family=processes, dimension=started
netdata_system_idlejitter_microseconds_lost_persec_average
    chart=system.idlejitter, family=idlejitter, dimension=average
    chart=system.idlejitter, family=idlejitter, dimension=max
    chart=system.idlejitter, family=idlejitter, dimension=min
netdata_system_interrupts_interrupts_persec_average
    chart=system.interrupts, family=interrupts, dimension=LOC
    chart=system.interrupts, family=interrupts, dimension=MCP
    chart=system.interrupts, family=interrupts, dimension=ata_piix_14
    chart=system.interrupts, family=interrupts, dimension=ata_piix_15
    chart=system.interrupts, family=interrupts, dimension=eth0_11
    chart=system.interrupts, family=interrupts, dimension=floppy_6
    chart=system.interrupts, family=interrupts, dimension=i8042_1
    chart=system.interrupts, family=interrupts, dimension=i8042_12
    chart=system.interrupts, family=interrupts, dimension=rtc0_8
    chart=system.interrupts, family=interrupts, dimension=serial_4
    chart=system.interrupts, family=interrupts, dimension=timer_0
netdata_system_intr_interrupts_persec_average
    chart=system.intr, family=interrupts, dimension=interrupts
netdata_system_io_KiB_persec_average
    chart=system.io, family=disk, dimension=in
    chart=system.io, family=disk, dimension=out
netdata_system_ip_kilobits_persec_average
    chart=system.ip, family=network, dimension=received
    chart=system.ip, family=network, dimension=sent
netdata_system_ipc_semaphore_arrays_arrays_average
    chart=system.ipc_semaphore_arrays, family=ipc semaphores, dimension=arrays
netdata_system_ipc_semaphores_semaphores_average
    chart=system.ipc_semaphores, family=ipc semaphores, dimension=semaphores
netdata_system_ipv6_kilobits_persec_average
    chart=system.ipv6, family=network, dimension=received
    chart=system.ipv6, family=network, dimension=sent
netdata_system_load_load_average
    chart=system.load, family=load, dimension=load1
    chart=system.load, family=load, dimension=load15
    chart=system.load, family=load, dimension=load5
netdata_system_net_kilobits_persec_average
    chart=system.net, family=network, dimension=received
    chart=system.net, family=network, dimension=sent
netdata_system_pgpgio_KiB_persec_average
    chart=system.pgpgio, family=disk, dimension=in
    chart=system.pgpgio, family=disk, dimension=out
netdata_system_processes_processes_average
    chart=system.processes, family=processes, dimension=blocked
    chart=system.processes, family=processes, dimension=running
netdata_system_ram_MiB_average
    chart=system.ram, family=ram, dimension=buffers
    chart=system.ram, family=ram, dimension=cached
    chart=system.ram, family=ram, dimension=free
    chart=system.ram, family=ram, dimension=used
```

```
netdata_system_shared_memory_bytes_bytes_average
    chart=system.shared_memory_bytes, family=ipc shared memory, dimension=bytes
netdata_system_shared_memory_segments_segments_average
    chart=system.shared_memory_segments, family=ipc shared memory, dimension=segments
netdata_system_softirqs_softirqs_persec_average
    chart=system.softirqs, family=softirqs, dimension=BLOCK
    chart=system.softirqs, family=softirqs, dimension=HRTIMER
    chart=system.softirqs, family=softirqs, dimension=NET_RX
    chart=system.softirqs, family=softirqs, dimension=NET_TX
    chart=system.softirqs, family=softirqs, dimension=RCU
    chart=system.softirqs, family=softirqs, dimension=TASKLET
    chart=system.softirqs, family=softirqs, dimension=TIMER
netdata_system_softnet_stat_events_persec_average
    chart=system.softnet_stat, family=softnet_stat, dimension=dropped
    chart=system.softnet_stat, family=softnet_stat, dimension=flow_limit_count
    chart=system.softnet_stat, family=softnet_stat, dimension=processed
    chart=system.softnet_stat, family=softnet_stat, dimension=received_rps
    chart=system.softnet_stat, family=softnet_stat, dimension=squeezed
netdata_system_swap_MiB_average
    chart=system.swap, family=swap, dimension=free
    chart=system.swap, family=swap, dimension=used
netdata_system_uptime_seconds_average
    chart=system.uptime, family=uptime, dimension=uptime
netdata_users_cpu_percentage_average
    chart=users.cpu, family=cpu, dimension=daemon
    chart=users.cpu, family=cpu, dimension=fields
    chart=users.cpu, family=cpu, dimension=messagebus
    chart=users.cpu, family=cpu, dimension=netdata
    chart=users.cpu, family=cpu, dimension=root
    chart=users.cpu, family=cpu, dimension=smmsp
netdata_users_cpu_system_percentage_average
    chart=users.cpu_system, family=cpu, dimension=daemon
    chart=users.cpu_system, family=cpu, dimension=fields
    chart=users.cpu_system, family=cpu, dimension=messagebus
    chart=users.cpu_system, family=cpu, dimension=netdata
    chart=users.cpu_system, family=cpu, dimension=root
    chart=users.cpu_system, family=cpu, dimension=smmsp
netdata_users_cpu_user_percentage_average
    chart=users.cpu_user, family=cpu, dimension=daemon
    chart=users.cpu_user, family=cpu, dimension=fields
    chart=users.cpu_user, family=cpu, dimension=messagebus
    chart=users.cpu_user, family=cpu, dimension=netdata
    chart=users.cpu_user, family=cpu, dimension=root
    chart=users.cpu_user, family=cpu, dimension=smmsp
netdata_users_files_open_files_average
    chart=users.files, family=disk, dimension=daemon
    chart=users.files, family=disk, dimension=fields
    chart=users.files, family=disk, dimension=messagebus
    chart=users.files, family=disk, dimension=netdata
    chart=users.files, family=disk, dimension=root
    chart=users.files, family=disk, dimension=smmsp
netdata_users_lreads_KiB_persec_average
    chart=users.lreads, family=disk, dimension=daemon
    chart=users.lreads, family=disk, dimension=fields
    chart=users.lreads, family=disk, dimension=messagebus
    chart=users.lreads, family=disk, dimension=netdata
    chart=users.lreads, family=disk, dimension=root
    chart=users.lreads, family=disk, dimension=smmsp
netdata_users_lwrites_KiB_persec_average
    chart=users.lwrites, family=disk, dimension=daemon
    chart=users.lwrites, family=disk, dimension=fields
    chart=users.lwrites, family=disk, dimension=messagebus
    chart=users.lwrites, family=disk, dimension=netdata
    chart=users.lwrites, family=disk, dimension=root
    chart=users.lwrites, family=disk, dimension=smmsp
netdata_users_major_faults_page_faults_persec_average
    chart=users.major_faults, family=swap, dimension=daemon
    chart=users.major_faults, family=swap, dimension=fields
    chart=users.major_faults, family=swap, dimension=messagebus
    chart=users.major_faults, family=swap, dimension=netdata
    chart=users.major_faults, family=swap, dimension=root
    chart=users.major_faults, family=swap, dimension=smmsp
netdata_users_mem_MiB_average
    chart=users.mem, family=mem, dimension=daemon
    chart=users.mem, family=mem, dimension=fields
    chart=users.mem, family=mem, dimension=messagebus
    chart=users.mem, family=mem, dimension=netdata
    chart=users.mem, family=mem, dimension=root
    chart=users.mem, family=mem, dimension=smmsp
netdata_users_minor_faults_page_faults_persec_average
    chart=users.minor_faults, family=mem, dimension=daemon
    chart=users.minor_faults, family=mem, dimension=fields
    chart=users.minor_faults, family=mem, dimension=messagebus
    chart=users.minor_faults, family=mem, dimension=netdata
    chart=users.minor_faults, family=mem, dimension=root
    chart=users.minor_faults, family=mem, dimension=smmsp
netdata_users_pipes_open_pipes_average
    chart=users.pipes, family=processes, dimension=daemon
    chart=users.pipes, family=processes, dimension=fields
    chart=users.pipes, family=processes, dimension=messagebus
    chart=users.pipes, family=processes, dimension=netdata
    chart=users.pipes, family=processes, dimension=root
    chart=users.pipes, family=processes, dimension=smmsp
netdata_users_preads_KiB_persec_average
    chart=users.preads, family=disk, dimension=daemon
    chart=users.preads, family=disk, dimension=fields
    chart=users.preads, family=disk, dimension=messagebus
    chart=users.preads, family=disk, dimension=netdata
    chart=users.preads, family=disk, dimension=root
    chart=users.preads, family=disk, dimension=smmsp
netdata_users_processes_processes_average
    chart=users.processes, family=processes, dimension=daemon
    chart=users.processes, family=processes, dimension=fields
    chart=users.processes, family=processes, dimension=messagebus
    chart=users.processes, family=processes, dimension=netdata
    chart=users.processes, family=processes, dimension=root
    chart=users.processes, family=processes, dimension=smmsp
netdata_users_pwrites_KiB_persec_average
    chart=users.pwrites, family=disk, dimension=daemon
    chart=users.pwrites, family=disk, dimension=fields
    chart=users.pwrites, family=disk, dimension=messagebus
    chart=users.pwrites, family=disk, dimension=netdata
    chart=users.pwrites, family=disk, dimension=root
    chart=users.pwrites, family=disk, dimension=smmsp
netdata_users_sockets_open_sockets_average
    chart=users.sockets, family=net, dimension=daemon
    chart=users.sockets, family=net, dimension=fields
    chart=users.sockets, family=net, dimension=messagebus
    chart=users.sockets, family=net, dimension=netdata
    chart=users.sockets, family=net, dimension=root
    chart=users.sockets, family=net, dimension=smmsp
netdata_users_swap_MiB_average
    chart=users.swap, family=swap, dimension=daemon
    chart=users.swap, family=swap, dimension=fields
    chart=users.swap, family=swap, dimension=messagebus
    chart=users.swap, family=swap, dimension=netdata
    chart=users.swap, family=swap, dimension=root
    chart=users.swap, family=swap, dimension=smmsp
netdata_users_threads_threads_average
    chart=users.threads, family=processes, dimension=daemon
    chart=users.threads, family=processes, dimension=fields
    chart=users.threads, family=processes, dimension=messagebus
    chart=users.threads, family=processes, dimension=netdata
    chart=users.threads, family=processes, dimension=root
    chart=users.threads, family=processes, dimension=smmsp
netdata_users_uptime_avg_seconds_average
    chart=users.uptime_avg, family=processes, dimension=daemon
    chart=users.uptime_avg, family=processes, dimension=fields
    chart=users.uptime_avg, family=processes, dimension=messagebus
    chart=users.uptime_avg, family=processes, dimension=netdata
    chart=users.uptime_avg, family=processes, dimension=root
    chart=users.uptime_avg, family=processes, dimension=smmsp
netdata_users_uptime_max_seconds_average
    chart=users.uptime_max, family=processes, dimension=daemon
    chart=users.uptime_max, family=processes, dimension=fields
    chart=users.uptime_max, family=processes, dimension=messagebus
    chart=users.uptime_max, family=processes, dimension=netdata
    chart=users.uptime_max, family=processes, dimension=root
    chart=users.uptime_max, family=processes, dimension=smmsp
netdata_users_uptime_min_seconds_average
    chart=users.uptime_min, family=processes, dimension=daemon
    chart=users.uptime_min, family=processes, dimension=fields
    chart=users.uptime_min, family=processes, dimension=messagebus
    chart=users.uptime_min, family=processes, dimension=netdata
    chart=users.uptime_min, family=processes, dimension=root
    chart=users.uptime_min, family=processes, dimension=smmsp
netdata_users_uptime_seconds_average
    chart=users.uptime, family=processes, dimension=daemon
    chart=users.uptime, family=processes, dimension=fields
    chart=users.uptime, family=processes, dimension=messagebus
    chart=users.uptime, family=processes, dimension=netdata
    chart=users.uptime, family=processes, dimension=root
    chart=users.uptime, family=processes, dimension=smmsp
netdata_users_vmem_MiB_average
    chart=users.vmem, family=mem, dimension=daemon
    chart=users.vmem, family=mem, dimension=fields
    chart=users.vmem, family=mem, dimension=messagebus
    chart=users.vmem, family=mem, dimension=netdata
    chart=users.vmem, family=mem, dimension=root
    chart=users.vmem, family=mem, dimension=smmsp
```

Appendix C: Netdata Selected Metrics Set

The subset of metrics collected by Netdata that were used to train the models for the Linux OS:

```
netdata_apps_cpu_percentage_average
    chart=apps.cpu, family=cpu, dimension=apps.plugin
    chart=apps.cpu, family=cpu, dimension=cron
    chart=apps.cpu, family=cpu, dimension=go.d.plugin
    chart=apps.cpu, family=cpu, dimension=kernel
    chart=apps.cpu, family=cpu, dimension=logs
    chart=apps.cpu, family=cpu, dimension='netdata
    chart=apps.cpu, family=cpu, dimension=other
    chart=apps.cpu, family=cpu, dimension=ssh
    chart=apps.cpu, family=cpu, dimension=system
    chart=apps.cpu, family=cpu, dimension=tc-qos-helper
netdata_apps_cpu_system_percentage_average
    chart=apps.cpu_system, family=cpu, dimension=apps.plugin
    chart=apps.cpu_system, family=cpu, dimension=cron
    chart=apps.cpu_system, family=cpu, dimension=go.d.plugin
    chart=apps.cpu_system, family=cpu, dimension=kernel
    chart=apps.cpu_system, family=cpu, dimension=logs
    chart=apps.cpu_system, family=cpu, dimension='netdata
    chart=apps.cpu_system, family=cpu, dimension=other
    chart=apps.cpu_system, family=cpu, dimension=ssh
    chart=apps.cpu_system, family=cpu, dimension=system
    chart=apps.cpu_system, family=cpu, dimension=tc-qos-helper
netdata_apps_cpu_user_percentage_average
    chart=apps.cpu_user, family=cpu, dimension=apps.plugin
    chart=apps.cpu_user, family=cpu, dimension=cron
    chart=apps.cpu_user, family=cpu, dimension=go.d.plugin
    chart=apps.cpu_user, family=cpu, dimension=logs
    chart=apps.cpu_user, family=cpu, dimension='netdata
    chart=apps.cpu_user, family=cpu, dimension=other
    chart=apps.cpu_user, family=cpu, dimension=ssh
    chart=apps.cpu_user, family=cpu, dimension=system
    chart=apps.cpu_user, family=cpu, dimension=tc-qos-helper
netdata_apps_files_open_files_average
    chart=apps.files, family=disk, dimension=apps.plugin
    chart=apps.files, family=disk, dimension=cron
    chart=apps.files, family=disk, dimension=go.d.plugin
    chart=apps.files, family=disk, dimension=logs
    chart=apps.files, family=disk, dimension='netdata
    chart=apps.files, family=disk, dimension=other
    chart=apps.files, family=disk, dimension=ssh
    chart=apps.files, family=disk, dimension=system
    chart=apps.files, family=disk, dimension=tc-qos-helper
netdata_apps_lreads_KiB_persec_average
    chart=apps.lreads, family=disk, dimension=apps.plugin
    chart=apps.lreads, family=disk, dimension=cron
    chart=apps.lreads, family=disk, dimension=logs
    chart=apps.lreads, family=disk, dimension='netdata
    chart=apps.lreads, family=disk, dimension=other
    chart=apps.lreads, family=disk, dimension=ssh
    chart=apps.lreads, family=disk, dimension=system
    chart=apps.lreads, family=disk, dimension=tc-qos-helper
netdata_apps_lwrites_KiB_persec_average
    chart=apps.lwrites, family=disk, dimension=apps.plugin
    chart=apps.lwrites, family=disk, dimension=cron
    chart=apps.lwrites, family=disk, dimension=go.d.plugin
    chart=apps.lwrites, family=disk, dimension=logs
    chart=apps.lwrites, family=disk, dimension='netdata
    chart=apps.lwrites, family=disk, dimension=other
    chart=apps.lwrites, family=disk, dimension=ssh
    chart=apps.lwrites, family=disk, dimension=system
    chart=apps.lwrites, family=disk, dimension=tc-qos-helper
netdata_apps_major_faults_page_faults_persec_average
    chart=apps.major_faults, family=swap, dimension='netdata
    chart=apps.major_faults, family=swap, dimension=other
    chart=apps.major_faults, family=swap, dimension=ssh
netdata_apps_mem_MiB_average
    chart=apps.mem, family=mem, dimension=apps.plugin
    chart=apps.mem, family=mem, dimension=cron
    chart=apps.mem, family=mem, dimension=dhcp
    chart=apps.mem, family=mem, dimension=go.d.plugin
    chart=apps.mem, family=mem, dimension=logs
    chart=apps.mem, family=mem, dimension='netdata
    chart=apps.mem, family=mem, dimension=other
    chart=apps.mem, family=mem, dimension=ssh
    chart=apps.mem, family=mem, dimension=system
    chart=apps.mem, family=mem, dimension=tc-qos-helper
netdata_apps_minor_faults_page_faults_persec_average
    chart=apps.minor_faults, family=mem, dimension=apps.plugin
    chart=apps.minor_faults, family=mem, dimension=cron
    chart=apps.minor_faults, family=mem, dimension=go.d.plugin
    chart=apps.minor_faults, family=mem, dimension=logs
    chart=apps.minor_faults, family=mem, dimension='netdata
    chart=apps.minor_faults, family=mem, dimension=other
    chart=apps.minor_faults, family=mem, dimension=ssh
    chart=apps.minor_faults, family=mem, dimension=system
    chart=apps.minor_faults, family=mem, dimension=tc-qos-helper
netdata_apps_pipes_open_pipes_average
    chart=apps.pipes, family=processes, dimension=apps.plugin
    chart=apps.pipes, family=processes, dimension=cron
    chart=apps.pipes, family=processes, dimension=go.d.plugin
    chart=apps.pipes, family=processes, dimension='netdata
    chart=apps.pipes, family=processes, dimension=other
    chart=apps.pipes, family=processes, dimension=ssh
    chart=apps.pipes, family=processes, dimension=tc-qos-helper
netdata_apps_preads_KiB_persec_average
    chart=apps.preads, family=disk, dimension=cron
    chart=apps.preads, family=disk, dimension='netdata
    chart=apps.preads, family=disk, dimension=other
    chart=apps.preads, family=disk, dimension=ssh
netdata_apps_processes_processes_average
    chart=apps.processes, family=processes, dimension=cron
    chart=apps.processes, family=processes, dimension=kernel
    chart=apps.processes, family=processes, dimension='netdata
    chart=apps.processes, family=processes, dimension=other
    chart=apps.processes, family=processes, dimension=ssh
    chart=apps.processes, family=processes, dimension=tc-qos-helper
netdata_apps_pwrites_KiB_persec_average
    chart=apps.pwrites, family=disk, dimension=apps.plugin
    chart=apps.pwrites, family=disk, dimension=cron
    chart=apps.pwrites, family=disk, dimension=go.d.plugin
    chart=apps.pwrites, family=disk, dimension=kernel
    chart=apps.pwrites, family=disk, dimension=logs
    chart=apps.pwrites, family=disk, dimension='netdata
    chart=apps.pwrites, family=disk, dimension=other
netdata_apps_sockets_open_sockets_average
    chart=apps.sockets, family=net, dimension=apps.plugin
    chart=apps.sockets, family=net, dimension=cron
    chart=apps.sockets, family=net, dimension=go.d.plugin
    chart=apps.sockets, family=net, dimension='netdata
    chart=apps.sockets, family=net, dimension=other
    chart=apps.sockets, family=net, dimension=ssh
    chart=apps.sockets, family=net, dimension=system
    chart=apps.sockets, family=net, dimension=tc-qos-helper
netdata_apps_threads_threads_average
    chart=apps.threads, family=processes, dimension=cron
    chart=apps.threads, family=processes, dimension=go.d.plugin
    chart=apps.threads, family=processes, dimension=kernel
    chart=apps.threads, family=processes, dimension='netdata
    chart=apps.threads, family=processes, dimension=other
    chart=apps.threads, family=processes, dimension=ssh
    chart=apps.threads, family=processes, dimension=tc-qos-helper
netdata_apps_vmem_MiB_average
    chart=apps.vmem, family=mem, dimension=apps.plugin
    chart=apps.vmem, family=mem, dimension=cron
    chart=apps.vmem, family=mem, dimension=logs
    chart=apps.vmem, family=mem, dimension='netdata
    chart=apps.vmem, family=mem, dimension=other
    chart=apps.vmem, family=mem, dimension=ssh
    chart=apps.vmem, family=mem, dimension=system
    chart=apps.vmem, family=mem, dimension=tc-qos-helper
netdata_cpu_cpu_percentage_average
    chart=cpu.cpu0, family=utilization, dimension=idle
    chart=cpu.cpu0, family=utilization, dimension=iowait
    chart=cpu.cpu0, family=utilization, dimension=irq
    chart=cpu.cpu0, family=utilization, dimension=softirq
    chart=cpu.cpu0, family=utilization, dimension=system
    chart=cpu.cpu0, family=utilization, dimension=user
netdata_cpu_interrupts_interrupts_persec_average
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=LOC
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=MCP
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=ata_piix_14
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=ata_piix_15
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=eth0_11
    chart=cpu.cpu0_interrupts, family=interrupts, dimension=serial_4
netdata_cpu_softnet_stat_events_persec_average
    chart=cpu.cpu0_softnet_stat, family=softnet_stat, dimension=processed
netdata_disk_avgsz_KiB_operation_average
    chart=disk_avgsz.sda, family=sda, dimension=reads
    chart=disk_avgsz.sda, family=sda, dimension=writes
netdata_disk_await_milliseconds_operation_average
    chart=disk_await.sda, family=sda, dimension=reads
    chart=disk_await.sda, family=sda, dimension=writes
netdata_disk_backlog_milliseconds_average
    chart=disk_backlog.sda, family=sda, dimension=backlog
netdata_disk_inodes_inodes_average
    chart=disk_inodes._, family=/, dimension=avail
    chart=disk_inodes._, family=/, dimension=used
    chart=disk_inodes._run, family=/run, dimension=avail
    chart=disk_inodes._run, family=/run, dimension=used
netdata_disk_io_KiB_persec_average
    chart=disk.sda, family=sda, dimension=reads
    chart=disk.sda, family=sda, dimension=writes
netdata_disk_iotime_milliseconds_persec_average
    chart=disk_iotime.sda, family=sda, dimension=reads
    chart=disk_iotime.sda, family=sda, dimension=writes
netdata_disk_mops_merged_operations_persec_average
    chart=disk_mops.sda, family=sda, dimension=reads
    chart=disk_mops.sda, family=sda, dimension=writes
netdata_disk_ops_operations_persec_average
    chart=disk_ops.sda, family=sda, dimension=reads
    chart=disk_ops.sda, family=sda, dimension=writes
```

```
netdata_disk_space_GiB_average
    chart=disk_space._, family=/, dimension=avail
    chart=disk_space._, family=/, dimension=used
    chart=disk_space._run, family=/run, dimension=avail
    chart=disk_space._run, family=/run, dimension=used
netdata_disk_svctm_milliseconds_operation_average
    chart=disk_svctm.sda, family=sda, dimension=svctm
netdata_disk_util___of_time_working_average
    chart=disk_util.sda, family=sda, dimension=utilization
netdata_groups_cpu_percentage_average
    chart=groups.cpu, family=cpu, dimension=fields
    chart=groups.cpu, family=cpu, dimension=messagebus
    chart=groups.cpu, family=cpu, dimension='netdata
    chart=groups.cpu, family=cpu, dimension=root
    chart=groups.cpu, family=cpu, dimension=smmsp
netdata_groups_cpu_system_percentage_average
    chart=groups.cpu_system, family=cpu, dimension=fields
    chart=groups.cpu_system, family=cpu, dimension=messagebus
    chart=groups.cpu_system, family=cpu, dimension='netdata
    chart=groups.cpu_system, family=cpu, dimension=root
    chart=groups.cpu_system, family=cpu, dimension=smmsp
netdata_groups_cpu_user_percentage_average
    chart=groups.cpu_user, family=cpu, dimension=fields
    chart=groups.cpu_user, family=cpu, dimension=messagebus
    chart=groups.cpu_user, family=cpu, dimension='netdata
    chart=groups.cpu_user, family=cpu, dimension=root
    chart=groups.cpu_user, family=cpu, dimension=smmsp
netdata_groups_files_open_files_average
    chart=groups.files, family=disk, dimension=fields
    chart=groups.files, family=disk, dimension='netdata
    chart=groups.files, family=disk, dimension=root
    chart=groups.files, family=disk, dimension=smmsp
netdata_groups_lreads_KiB_persec_average
    chart=groups.lreads, family=disk, dimension=fields
    chart=groups.lreads, family=disk, dimension=messagebus
    chart=groups.lreads, family=disk, dimension='netdata
    chart=groups.lreads, family=disk, dimension=root
    chart=groups.lreads, family=disk, dimension=smmsp
netdata_groups_lwrites_KiB_persec_average
    chart=groups.lwrites, family=disk, dimension=fields
    chart=groups.lwrites, family=disk, dimension='netdata
    chart=groups.lwrites, family=disk, dimension=root
    chart=groups.lwrites, family=disk, dimension=smmsp
netdata_groups_major_faults_page_faults_persec_average
    chart=groups.major_faults, family=swap, dimension=fields
    chart=groups.major_faults, family=swap, dimension='netdata
    chart=groups.major_faults, family=swap, dimension=root
netdata_groups_mem_MiB_average
    chart=groups.mem, family=mem, dimension=daemon
    chart=groups.mem, family=mem, dimension=fields
    chart=groups.mem, family=mem, dimension=messagebus
    chart=groups.mem, family=mem, dimension='netdata
    chart=groups.mem, family=mem, dimension=root
    chart=groups.mem, family=mem, dimension=smmsp
netdata_groups_minor_faults_page_faults_persec_average
    chart=groups.minor_faults, family=mem, dimension=fields
    chart=groups.minor_faults, family=mem, dimension=messagebus
    chart=groups.minor_faults, family=mem, dimension='netdata
    chart=groups.minor_faults, family=mem, dimension=root
    chart=groups.minor_faults, family=mem, dimension=smmsp
netdata_groups_pipes_open_pipes_average
    chart=groups.pipes, family=processes, dimension=fields
    chart=groups.pipes, family=processes, dimension='netdata
    chart=groups.pipes, family=processes, dimension=root
    chart=groups.pipes, family=processes, dimension=smmsp
netdata_groups_preads_KiB_persec_average
    chart=groups.preads, family=disk, dimension=fields
    chart=groups.preads, family=disk, dimension='netdata
    chart=groups.preads, family=disk, dimension=root
    chart=groups.preads, family=disk, dimension=smmsp
netdata_groups_processes_processes_average
    chart=groups.processes, family=processes, dimension=fields
    chart=groups.processes, family=processes, dimension='netdata
    chart=groups.processes, family=processes, dimension=root
    chart=groups.processes, family=processes, dimension=smmsp
netdata_groups_pwrites_KiB_persec_average
    chart=groups.pwrites, family=disk, dimension=fields
    chart=groups.pwrites, family=disk, dimension='netdata
    chart=groups.pwrites, family=disk, dimension=root
    chart=groups.pwrites, family=disk, dimension=smmsp
netdata_groups_sockets_open_sockets_average
    chart=groups.sockets, family=net, dimension=fields
    chart=groups.sockets, family=net, dimension=messagebus
    chart=groups.sockets, family=net, dimension='netdata
    chart=groups.sockets, family=net, dimension=root
    chart=groups.sockets, family=net, dimension=smmsp
netdata_groups_threads_threads_average
    chart=groups.threads, family=processes, dimension=fields
    chart=groups.threads, family=processes, dimension=’netdata
    chart=groups.threads, family=processes, dimension=root
    chart=groups.threads, family=processes, dimension=smmsp
netdata_groups_vmem_MiB_average
    chart=groups.vmem, family=mem, dimension=fields
    chart=groups.vmem, family=mem, dimension=’netdata
    chart=groups.vmem, family=mem, dimension=root
    chart=groups.vmem, family=mem, dimension=smmsp
netdata_ip_ecnpkts_packets_persec_average
    chart=ip.ecnpkts, family=ecn, dimension=NoECTP
netdata_ipv4_packets_packets_persec_average
    chart=ipv4.packets, family=packets, dimension=delivered
    chart=ipv4.packets, family=packets, dimension=received
    chart=ipv4.packets, family=packets, dimension=sent
netdata_ipv4_sockstat_sockets_sockets_average
    chart=ipv4.sockstat_sockets, family=sockets, dimension=used
netdata_ipv4_sockstat_tcp_mem_KiB_average
    chart=ipv4.sockstat_tcp_mem, family=tcp, dimension=mem
netdata_ipv4_sockstat_tcp_sockets_sockets_average
    chart=ipv4.sockstat_tcp_sockets, family=tcp, dimension=alloc
    chart=ipv4.sockstat_tcp_sockets, family=tcp, dimension=inuse
    chart=ipv4.sockstat_tcp_sockets, family=tcp, dimension=orphan
    chart=ipv4.sockstat_tcp_sockets, family=tcp, dimension=timewait
netdata_ipv4_sockstat_udp_sockets_sockets_average
    chart=ipv4.sockstat_udp_sockets, family=udp, dimension=inuse
netdata_ipv4_tcphandshake_events_persec_average
    chart=ipv4.tcphandshake, family=tcp, dimension=AttemptFails
    chart=ipv4.tcphandshake, family=tcp, dimension=EstabResets
    chart=ipv4.tcphandshake, family=tcp, dimension=OutRsts
netdata_ipv4_tcpopens_connections_persec_average
    chart=ipv4.tcpopens, family=tcp, dimension=active
    chart=ipv4.tcpopens, family=tcp, dimension=passive
netdata_ipv4_tcppackets_packets_persec_average
    chart=ipv4.tcppackets, family=tcp, dimension=received
    chart=ipv4.tcppackets, family=tcp, dimension=sent
netdata_ipv4_tcpsock_active_connections_average
    chart=ipv4.tcpsock, family=tcp, dimension=connections
netdata_ipv4_udppackets_packets_persec_average
    chart=ipv4.udppackets, family=udp, dimension=received
    chart=ipv4.udppackets, family=udp, dimension=sent
netdata_ipv6_ect_packets_persec_average
    chart=ipv6.ect, family=packets, dimension=InNoECTPkts
netdata_ipv6_icmp_messages_persec_average
    chart=ipv6.icmp, family=icmp6, dimension=received
netdata_ipv6_icmprouter_messages_persec_average
    chart=ipv6.icmprouter, family=icmp6, dimension=InAdvertisements
netdata_ipv6_mcast_kilobits_persec_average
    chart=ipv6.mcast, family=multicast6, dimension=received
netdata_ipv6_mcastpkts_packets_persec_average
    chart=ipv6.mcastpkts, family=multicast6, dimension=received
netdata_ipv6_packets_packets_persec_average
    chart=ipv6.packets, family=packets, dimension=delivers
    chart=ipv6.packets, family=packets, dimension=received
netdata_ipv6_sockstat6_tcp_sockets_sockets_average
    chart=ipv6.sockstat6_tcp_sockets, family=tcp6, dimension=inuse
netdata_ipv6_sockstat6_udp_sockets_sockets_average
    chart=ipv6.sockstat6_udp_sockets, family=udp6, dimension=inuse
netdata_mem_available_MiB_average
    chart=mem.available, family=system, dimension=avail
netdata_mem_committed_MiB_average
    chart=mem.committed, family=system, dimension=Committed_AS
netdata_mem_kernel_MiB_average
    chart=mem.kernel, family=kernel, dimension=KernelStack
    chart=mem.kernel, family=kernel, dimension=PageTables
    chart=mem.kernel, family=kernel, dimension=Slab
    chart=mem.kernel, family=kernel, dimension=VmallocUsed
netdata_mem_pgfaults_faults_persec_average
    chart=mem.pgfaults, family=system, dimension=major
    chart=mem.pgfaults, family=system, dimension=minor
netdata_mem_slab_MiB_average
    chart=mem.slab, family=slab, dimension=reclaimable
    chart=mem.slab, family=slab, dimension=unreclaimable
netdata_mem_writeback_MiB_average
    chart=mem.writeback, family=kernel, dimension=Dirty
    chart=mem.writeback, family=kernel, dimension=Writeback
netdata_net_errors_errors_persec_average
    chart=net_errors.eth0, family=eth0, dimension=inbound
netdata_net_events_events_persec_average
    chart=net_events.eth0, family=eth0, dimension=frames
netdata_net_net_kilobits_persec_average
    chart=net.eth0, family=eth0, dimension=received
    chart=net.eth0, family=eth0, dimension=sent
```

```
netdata_net_packets_packets_persec_average
    chart=net_packets.eth0, family=eth0, dimension=received
    chart=net_packets.eth0, family=eth0, dimension=sent
netdata_system_active_processes_processes_average
    chart=system.active_processes, family=processes, dimension=active
netdata_system_cpu_percentage_average
    chart=system.cpu, family=cpu, dimension=idle
    chart=system.cpu, family=cpu, dimension=iowait
    chart=system.cpu, family=cpu, dimension=irq
    chart=system.cpu, family=cpu, dimension=softirq
    chart=system.cpu, family=cpu, dimension=system
    chart=system.cpu, family=cpu, dimension=user
netdata_system_ctxt_context_switches_persec_average
    chart=system.ctxt, family=processes, dimension=switches
netdata_system_entropy_entropy_average
    chart=system.entropy, family=entropy, dimension=entropy
netdata_system_forks_processes_persec_average
    chart=system.forks, family=processes, dimension=started
netdata_system_idlejitter_microseconds_lost_persec_average
    chart=system.idlejitter, family=idlejitter, dimension=average
    chart=system.idlejitter, family=idlejitter, dimension=max
    chart=system.idlejitter, family=idlejitter, dimension=min
netdata_system_interrupts_interrupts_persec_average
    chart=system.interrupts, family=interrupts, dimension=LOC
    chart=system.interrupts, family=interrupts, dimension=MCP
    chart=system.interrupts, family=interrupts, dimension=ata_piix_14
    chart=system.interrupts, family=interrupts, dimension=ata_piix_15
    chart=system.interrupts, family=interrupts, dimension=eth0_11
    chart=system.interrupts, family=interrupts, dimension=serial_4
netdata_system_intr_interrupts_persec_average
    chart=system.intr, family=interrupts, dimension=interrupts
netdata_system_io_KiB_persec_average
    chart=system.io, family=disk, dimension=in
    chart=system.io, family=disk, dimension=out
netdata_system_ip_kilobits_persec_average
    chart=system.ip, family=network, dimension=received
    chart=system.ip, family=network, dimension=sent
netdata_system_ipv6_kilobits_persec_average
    chart=system.ipv6, family=network, dimension=received
netdata_system_net_kilobits_persec_average
    chart=system.net, family=network, dimension=received
    chart=system.net, family=network, dimension=sent
netdata_system_pgpgio_KiB_persec_average
    chart=system.pgpgio, family=disk, dimension=in
    chart=system.pgpgio, family=disk, dimension=out
netdata_system_processes_processes_average
    chart=system.processes, family=processes, dimension=blocked
    chart=system.processes, family=processes, dimension=running
netdata_system_ram_MiB_average
    chart=system.ram, family=ram, dimension=buffers
    chart=system.ram, family=ram, dimension=cached
    chart=system.ram, family=ram, dimension=free
    chart=system.ram, family=ram, dimension=used
netdata_system_softnet_stat_events_persec_average
    chart=system.softnet_stat, family=softnet_stat, dimension=processed
netdata_users_cpu_percentage_average
    chart=users.cpu, family=cpu, dimension=fields
    chart=users.cpu, family=cpu, dimension=messagebus
    chart=users.cpu, family=cpu, dimension='netdata
    chart=users.cpu, family=cpu, dimension=root
netdata_users_cpu_system_percentage_average
    chart=users.cpu_system, family=cpu, dimension=fields
    chart=users.cpu_system, family=cpu, dimension=messagebus
    chart=users.cpu_system, family=cpu, dimension='netdata
    chart=users.cpu_system, family=cpu, dimension=root
netdata_users_cpu_user_percentage_average
    chart=users.cpu_user, family=cpu, dimension=fields
    chart=users.cpu_user, family=cpu, dimension=messagebus
    chart=users.cpu_user, family=cpu, dimension='netdata
    chart=users.cpu_user, family=cpu, dimension=root
netdata_users_files_open_files_average
    chart=users.files, family=disk, dimension=fields
    chart=users.files, family=disk, dimension='netdata
    chart=users.files, family=disk, dimension=root
netdata_users_lreads_KiB_persec_average
    chart=users.lreads, family=disk, dimension=fields
    chart=users.lreads, family=disk, dimension=messagebus
    chart=users.lreads, family=disk, dimension='netdata
    chart=users.lreads, family=disk, dimension=root
netdata_users_lwrites_KiB_persec_average
    chart=users.lwrites, family=disk, dimension=fields
    chart=users.lwrites, family=disk, dimension='netdata
    chart=users.lwrites, family=disk, dimension=root
netdata_users_major_faults_page_faults_persec_average
    chart=users.major_faults, family=swap, dimension=fields
    chart=users.major_faults, family=swap, dimension='netdata
    chart=users.major_faults, family=swap, dimension=root
netdata_users_mem_MiB_average
    chart=users.mem, family=mem, dimension=daemon
    chart=users.mem, family=mem, dimension=fields
    chart=users.mem, family=mem, dimension=messagebus
    chart=users.mem, family=mem, dimension='netdata
    chart=users.mem, family=mem, dimension=root
netdata_users_minor_faults_page_faults_persec_average
    chart=users.minor_faults, family=mem, dimension=fields
    chart=users.minor_faults, family=mem, dimension=messagebus
    chart=users.minor_faults, family=mem, dimension='netdata
    chart=users.minor_faults, family=mem, dimension=root
netdata_users_pipes_open_pipes_average
    chart=users.pipes, family=processes, dimension=fields
    chart=users.pipes, family=processes, dimension='netdata
    chart=users.pipes, family=processes, dimension=root
netdata_users_preads_KiB_persec_average
    chart=users.preads, family=disk, dimension=fields
    chart=users.preads, family=disk, dimension='netdata
    chart=users.preads, family=disk, dimension=root
netdata_users_processes_processes_average
    chart=users.processes, family=processes, dimension=fields
    chart=users.processes, family=processes, dimension='netdata
    chart=users.processes, family=processes, dimension=root
netdata_users_pwrites_KiB_persec_average
    chart=users.pwrites, family=disk, dimension=fields
    chart=users.pwrites, family=disk, dimension='netdata
    chart=users.pwrites, family=disk, dimension=root
netdata_users_sockets_open_sockets_average
    chart=users.sockets, family=net, dimension=fields
    chart=users.sockets, family=net, dimension=messagebus
    chart=users.sockets, family=net, dimension='netdata
    chart=users.sockets, family=net, dimension=root
netdata_users_threads_threads_average
    chart=users.threads, family=processes, dimension=fields
    chart=users.threads, family=processes, dimension='netdata
    chart=users.threads, family=processes, dimension=root
netdata_users_vmem_MiB_average
    chart=users.vmem, family=mem, dimension=fields
    chart=users.vmem, family=mem, dimension='netdata
    chart=users.vmem, family=mem, dimension=root
```

Appendix D: Failure Detectors

This section briefly details the failure detectors implemented for the fault injection campaigns. Several independent failure detectors were deployed, more precisely:

- 'failed_boot': control/address issues with boot, relaunch if necessary
- 'failed_fault_injection': fault injection was unsuccessful
- 'excessive_failures': too many failures (more than 30)
- 'crash': the OS has crashed or rebooted
- 'hang': the OS hangs; this is detected by not responding to ping, writing to disk, or accepting connections
- 'impaired_ssh': SSH not working as expected (e.g., no connection/transfer)
- 'unavailable_ssh': it was not possible to establish an SSH connection after 10 attempts
- 'impaired_io': the OS no longer writes to disk
- 'command_timeout': a given command is not executed or takes too long
- 'value': the performance of the workload is lower than the expected baseline
- 'fsck_data_corruption': the *fsck* tool detects a filesystem corruption

- ‘infinite_execution’: the workload execution takes 50% longer than expected
- ‘incomplete_execution’: the workload terminated earlier than expected
- ‘empty_benchmark_results’: the workload returned an empty result set
- ‘no_benchmark_results’: the workload did not return any results
- ‘corrupt_netdata_encoding’: netdata data encoding was corrupted
- ‘corrupt_watchdog_encoding’: watchdog data encoding was corrupted
- ‘corrupt_pexpect_encoding’: pexpect data encoding was corrupted
- ‘corrupt_logs_encoding’: logs data encoding was corrupted

The testbed also monitored the system logs (i.e., *dmesg.log, kern.log,* and the *tty* used to interact with the target machine) to detect non-fail-stop failures and/or complement the failure detectors previously described. As the goal was to catch as many potential failure messages as possible, some of the failures are specializations of a more generic version (e.g., *corrupt_kernel* is a specialization of the *bug_generic* failure). To avoid logging multiple failures for the same failure event, in such situations the parser would analyze the position of the failure in the logs and remove the generic failure. The following failures were monitored:

- ‘kernel_offset’
  match: ‘Kernel Offset’
- ‘panic’
  match: ‘kernel panic’
- ‘panic_interrupt’
  match: ‘Kernel panic - not syncing: Fatal exception in interrupt’
  overwrites: ‘panic’
- ‘resource_exhaustion’
  match: ‘killed’
- ‘shutdown_errors’
  match: ‘killed by TERM signal’
  overwrites: ‘resource_exhaustion’
- ‘memory_exhaustion’
  match: ‘Cannot allocate memory’
- ‘memory_corruption’
  match: ‘has RLIMIT_CORE set to 1’
- ‘failed_tests’
  match: ‘tests failed to properly run’
- ‘bug_generic’
  match: ‘BUG:’

- ‘corrupt_kernel’
  match: ‘BUG: unable to handle kernel’
  overwrites: ‘bug_generic’
- ‘corrupt_kernel_null_pointer’
  match: ‘BUG: unable to handle kernel null’
  overwrites: ‘corrupt_kernel, bug_generic’
- ‘corrupt_kernel_paging’
  match: ‘BUG: unable to handle kernel paging’
  overwrites: ‘corrupt_kernel, bug_generic’
- ‘error_generic’
  match: ‘ERROR:’
- ‘warning_generic’
  match: ‘WARNING:’
- ‘notice_generic’
  match: ‘NOTICE:’
- ‘kernel_bug’
  match: ‘Kernel BUG’
- ‘corrupt_filesystem_structure’
  match: ‘cd to /usr/share/phoronix-test-suite’
  match: ‘directory nonexistent’
  match: ‘command not found’
  match: ‘Invalid argument’
- ‘kernel_stack_corruption’
  match: ‘stack is corrupted’
- ‘cpu_lock’
  match: ‘BUG: soft lockup’
  overwrites: ‘bug_generic’
- ‘exploit_attempt’
  match: ‘protected page’
- ‘recursive_fault’
  match: ‘recursive fault but reboot is needed’
- ‘import_error’
  match: ‘ImportError:’
- ‘no_space_left’
  match: ‘Cannot mmap to shared memory’
- ‘segmentation_fault’

match: ‘Segmentation fault’
match: ‘segfault’

- ‘nohz’
  match: ‘NOHZ:’
- ‘invalid_opcode’
  match: ‘invalid opcode’
- ‘divide_error’
  match: ‘divide error’
- ‘blocked_execution’
  match: ‘blocked for more than’
- ‘io_error’
  match: ‘I/O error’
- ‘long_argument_list’
  match: ‘Argument list too long’
- ‘too_many_open_files’
  match: ‘Too many open files’
- ‘out_of_memory’
  match: ‘Out of memory’
- ‘exception_emask’
  match: ‘exception Emask’
- ‘error_shared_libraries’
  match: ‘error while loading shared libraries’
- ‘bad_page_map’
  match: ‘BUG: Bad page map’
  overwrites: ‘bug_generic’
- ‘bad_rss_counter’
  match: ‘BUG: Bad rss-counter state’
  overwrites: ‘bug_generic’
- ‘corrupt_page_table’
  match: ‘Corrupted page table’
- ‘corrupt_journal’
  match: ‘Detected aborted journal’
- ‘ext4_fs’
  match: ‘EXT4-fs error’
- ‘input_output_error’

match: ‘Input/output error’

- ‘corrupt_filesystem’
  match: ‘delayed block allocation’
- ‘corrupt_system’
  match: ‘ERROR: In procedure’
  match: ‘unable to load plugins’
  overwrites: ‘error_generic’
- ‘corrupt_fork’
  match: ‘fork: retry: No child processes’
  match: ‘Cannot fork’
- ‘unavailable_resource’
  match: ‘Resource temporarily unavailable’
- ‘rcu_stall’
  match: ‘rcu_sched detected stall’
  match: ‘rcu_sched self-detected stall’
- ‘aborting’
  match: ‘aborting’
- ‘general_fault’
  match: ‘general protection fault’
- ‘possible_corrupt_cifs’
  match: ‘CIFS VFS’
- ‘corrupt_cifs’
  match: ‘CIFS VFS: Send error in write’
  match: ‘CIFS VFS: tcp sent no data’
  match: ‘CIFS VFS: Illegal’
  match: ‘CIFS VFS: Error’
  match: ‘CIFS VFS: Bad’
  match: ‘CIFS VFS: No’
  match: ‘CIFS VFS: RFC’
  match: ‘Host is down’
  match: ‘cifs_vfs_err’
  overwrites: ‘possible_corrupt_cifs’
- ‘warning_ext4_evict_inode’
  match: ‘ext4_evict_inode’
- ‘warning_mmap’
  match: ‘WARNING: at mm/mmap’
  overwrites: ‘warning_generic’
- ‘warning_softirq’

match: ‘WARNING: at kernel/softirq’
overwrites: ‘warning_generic’

- ‘no_irq_handler’
  match: ‘No irq handler for vector’
- ‘corrupt_ata’
  match: ‘lost interrupt’
  match: ‘qc timeout’
- ‘page_allocation_failure’
  match: ‘page allocation failure’
- ‘cannot_read_realtime_clock’
  match: ‘cannot read realtime clock’
- ‘dma_map_failed’
  match: ‘TX DMA map failed’
- ‘nommu_map_single_overflow’
  match: ‘nommu_map_single: overflow’
- ‘stress_ng_unsuccessful_run’
  match: ‘unsuccessful run’
- ‘schedule_timeout’
  match: ‘schedule_timeout: wrong timeout’
- ‘unstable_clocksource’
  match: ‘Clocksource tsc unstable’
- ‘invalid_irq’
  match: ‘bogus return value’
- ‘corrupt_permissions’
  match: ‘Attempt to access syslog with CAP_SYS_ADMIN’
- ‘failed_spawn’
  match: ‘Failed to spawn’
- ‘bad_file_descriptor’
  match: ‘Bad file descriptor’
- ‘network_reset_adapter’
  match: ‘Reset adapter unexpectedly’
- ‘tx_unit_hang’
  match: ‘Detected Tx Unit Hang’
- ‘runaway_loop_modprobe’
  match: ‘runaway loop modprobe’

- ‘exec_format_error’
  match: ‘Exec format error’
- ‘no_vm86_info’
  match: ‘vm86_32: no vm86_info: BAD’
- ‘no_sysfs_cache’
  match: ‘cache allocate: using defaults, can’t determine cache details from sysfs’
- ‘invalid_softirq_preempt_count’
  match: ‘softirq: huh, entered softirq’
- ‘compromised_vm_tunneling’
  match: ‘Operation too slow. Less than’