AGH University

FIELD OF SCIENCE: Engineering and Technology

SCIENTIFIC DISCIPLINE: Information and Communication Technology

DOCTORAL THESIS

Automation of Anomaly Detection in Base Stations of Cellular Networks with Machine Learning

Author: Sebastian Zarębski

First supervisor: Piotr Chołda, Ph.D.
Assisting supervisor: Krzysztof Rusek, Ph.D.

Completed in:
AGH University of Krakow
Faculty of Computer Science, Electronics and Telecommunications
Institute of Telecommunications

Kraków, 2025

AGH University

DZIEDZINA Nauki Inżynieryjno-Techniczne

DYSCYPLINA WIODĄCA Informatyka Techniczna i Telekomunikacja

ROZPRAWA DOKTORSKA

Automatyzacja wykrywania anomalii w stacjach bazowych sieci komórkowych z użyciem uczenia maszynowego

Autor: Sebastian Zarębski

Promotor rozprawy: Dr hab. inż. Piotr Chołda, prof. AGH
Drugi promotor: Dr inż. Krzysztof Rusek

Praca wykonana:
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Informatyki, Elektroniki i Telekomunikacji
Instytut Telekomunikacji

Kraków, 2025

Acknowledgements

I would like to express my deepest gratitude to my beloved wife, Adrianna, for her unwavering support, patience, and encouragement. Your belief in me was my greatest motivation throughout this journey. I am also incredibly grateful for the love and understanding of my entire family.

I am deeply grateful to my supervisors, Piotr Chołda and Krzysztof Rusek, for their invaluable guidance and mentorship. Your insightful feedback was instrumental in shaping this work and helping me overcome numerous challenges.

I extend my special thanks to Ewa Marchewka, my company's supervisor, for her invaluable guidance and support. My special thanks also go to the team from the BRUNO project at NOKIA, whose collaboration was essential in bringing the industrial implementation of this research to life. Special thanks to Remigiusz Michułka, Martyna Świeściak, Jakub Feluś, Aleksander Kuźmich, Marcin Jokiej, Adrian Kubała, Sebastian Sitko.

Finally, I thank my friends and colleagues whose support and encouragement made this long journey much more enjoyable.

Abstract

The transition of cellular networks to 4G and 5G has introduced significant challenges due to the increased complexity of base station systems, particularly impacting software quality assurance. This dissertation, a result of an **industrial doctorate** conducted at **NOKIA**, focuses on addressing these challenges in software engineering. The work is specifically centered on automating processes within the pre-shipment phase of software testing, rather than on real-time, online anomaly detection in live networks. In this context, an anomaly is defined as any software defect, bug, or unexpected system behavior identified during testing prior to deployment. Traditional methods of software testing and anomaly management within continuous integration (CI) and continuous deployment (CD) pipelines have become increasingly inefficient and resource-intensive, increasing the likelihood of software problems that could disrupt critical network services. To address these challenges, this dissertation proposes and empirically validates two distinct but complementary machine learning (ML) frameworks aimed at improving the efficiency, automation, and reliability of anomaly detection processes, thus improving software quality assurance in the telecommunications sector.

The first major contribution targets test set optimization (TSO), especially in the highly demanding regression testing phase. To increase the efficiency of anomaly detection, the study presents **LMLDA**. By integrating data from software code modifications and textual descriptions of test cases, it presents an effective, resource-efficient alternative to large language models (LLM), demonstrating superior bug coverage over established baselines, and it also promotes automated selection processes for test sets, allowing significant time gains of reor-

ganizing the original test queue. This was accomplished with a new test sequence that prioritizes potentially problematic tests by moving them to the start. This reorder resulted in speeding up the discovery of key problems, reducing the time to bug discovery on average by 8 hours in daily routines, allowing for faster resolutions and reducing time-to-fix.

The second significant contribution addresses automated ticket routing of anomalies discovered with LMLDA. First, it involves setting up a pre-processing phase focused on augmenting comprehensive and specific data, particularly automating the detection and expansion of acronyms and abbreviations. Furthermore, due to potential overlaps in discoveries by global teams, a contingency plan is implemented to identify similar tickets and minimize false samples in a dataset with the Siamese neural network. Lastly, the system for classifying the anomalies to proper departments must deliver confident predictions or alert the engineer for manual review. To achieve it, this dissertation introduces a **two-stage Bayesian framework** designed to develop a reliable and trust-based classification system. Initially, this framework uses LoRA to fine-tune a LLM to extract domain-specific features, e.g. acronyms like Multi-Input Multi-Output (MIMO). Following this, a **Bayesian classification head** is trained to assess epistemic uncertainty, enabling the model to identify its own limitations. Comprehensive evaluations indicated that the foundational architecture of LLM, especially those incorporating various attention mechanisms, plays a crucial role in determining self-performance. Additionally, it was found that priors with lighter tails, such as the Laplace distribution, are more effective in this context compared to those with heavier tails. The developed human-in-the-loop (HITL) system is capable of automating nearly 40% of bug reports with a great precision of 98%, while effectively escalating low-confidence samples to human experts for manual expertise. Although 40% may seem modest compared to theoretical benchmarks often cited in machine learning research, in this industrial context, it represents a significant breakthrough. Automating nearly half of the bug reports, each of which typically requires several hours to days of root cause analysis, translates into substantial time and cost savings, enabling software developers to address issues more efficiently and accelerate the delivery of fixes.

By delivering two distinct, proprietary solutions, this dissertation provides

a blueprint for developing efficient and cost-effective automation in critical software engineering pipelines. It demonstrates a crucial shift from pursuing accuracy alone to building reliable, interpretable, and uncertainty-aware ML systems, improving the resilience and efficiency of software development in modern telecommunication networks.

Keywords: 5G, Anomaly Detection, Bayesian Neural Networks (BNN), Bug Triage, Epistemic Uncertainty, Human-in-the-Loop (HITL), Large Language Models (LLM), LMLDA, Machine Learning, Software Testing, Telecommunications, Test Set Optimization (TSO), Uncertainty Quantification

Streszczenie

Ewolucja sieci komórkowych w kierunku technologii 4G i 5G ujawniła wyzwania w ich tworzeniu, wynikające z rosnącej złożoności systemów stacji bazowych, co w szczególności wpłynęło na zapewnienie jakości ich oprogramowania. Niniejsza rozprawa doktorska, będąca rezultatem **doktoratu wdrożeniowego** zrealizowanego w firmie **NOKIA**, skupia się na wspomnianych wyzwaniach z obszaru inżynierii oprogramowania. Praca koncentruje się w szczególności na automatyzacji procesów w ramach przedwdrożeniowej fazy testowania oprogramowania, a nie na wykrywaniu anomalii w czasie rzeczywistym w działających sieciach. W tym ujęciu anomalia jest definiowana jako każda wada oprogramowania, błąd lub nieoczekiwane zachowanie systemu, zidentyfikowane podczas testów przed wdrożeniem produktu. Tradycyjne metody testowania oprogramowania i zarządzania anomaliami w ramach praktyki ciągłej integracji (CI) i ciągłego wdrażania (CD) stają się coraz mniej wydajne i zasobochłonne, co zwiększa prawdopodobieństwo wystąpienia problemów, które mogą zakłócić działanie kluczowych usług sieciowych. W odpowiedzi na te wyzwania, w rozprawie zaproponowano oraz empirycznie zweryfikowano dwa odrębne, lecz uzupełniające się systemy oparte na uczeniu maszynowym (ML), których celem jest zwiększenie wydajności, stopnia automatyzacji i niezawodności procesów wykrywania anomalii, a w konsekwencji usprawnienie procesów zapewnienia jakości oprogramowania w sektorze telekomunikacyjnym.

Pierwszy istotny wkład pracy dotyczy obszaru optymalizacji kolejki testów (TSO), zwłaszcza w odniesieniu do fazy testów regresyjnych. W celu zwiększenia skuteczności wykrywania anomalii, w pracy przedstawiono autorski sys-

tem **LMLDA**. System ten, integrując dane o modyfikacjach kodu z tekstowymi opisami przypadków testowych, stanowi skuteczną i zasobooszczędną alternatywę dla wielkich modeli językowych (LLM), wykazując przy tym lepsze pokrycie błędów w porównaniu do metod bazowych. Ponadto, wspiera on procesy automatycznego doboru zestawów testowych, co pozwala na znaczną oszczędność czasu dzięki reorganizacji pierwotnej kolejki testów. Osiągnięto to dzięki zmianie kolejności sekwencji testów, która nadaje priorytet testom o największym prawdopodobieństwie wykrycia błędu, przesuwając je na początek kolejki wykonywania. Zmiana ta przyspieszyła wykrywanie kluczowych problemów, skracając w codziennej pracy średni czas potrzebny na znalezienie błędu o 8 godzin, co pozwoliło na szybsze wdrażanie poprawek i skróciło całkowity czas naprawy.

Drugi znaczący wkład pracy dotyczy automatycznego kierowania zgłoszeń (tzw. ticketów) dotyczących anomalii wykrytych za pomocą systemu LMLDA. Proces ten obejmuje, w pierwszej kolejności, etap wstępnego przetwarzania danych, skoncentrowany na ich uzupełnianiu i uszczegóławianiu, zwłaszcza poprzez automatyczne wykrywanie i rozwijanie akronimów oraz skrótów. Ponadto, ze względu na potencjalne duplikowanie zgłoszeń przez międzynarodowe zespoły, wdrożono mechanizm oparty na syjamskiej sieci neuronowej, który identyfikuje podobne zgłoszenia i minimalizuje liczbę fałszywych próbek w zbiorze danych. Wreszcie, system klasyfikujący anomalie do odpowiednich zespołów musi dostarczać predykcji o wysokim stopniu pewności lub sygnalizować inżynierowi konieczność ręcznej weryfikacji. Aby to osiągnąć, w niniejszej rozprawie wprowadzono **dwuetapowy system Bayesowski**, zaprojektowany w celu stworzenia niezawodnego i opartego na zaufaniu systemu klasyfikacji. W pierwszym etapie, system ten wykorzystuje technikę LoRA do dostrojenia modelu LLM w celu wydobycia cech specyficznych dla danej dziedziny, takich jak akronimy, np. MIMO. Następnie, trenowana jest **Bayesowska głowica klasyfikacyjna** w celu oceny niepewności epistemicznej, co pozwala modelowi na identyfikację własnych ograniczeń. Badania wykazały, że podstawy architektury LLM, zwłaszcza modeli wykorzystujących różne mechanizmy uwagi (z ang. *attention*), odgrywają kluczową rolę w zdolności systemu do oceny własnej skuteczności. Dodatkowo stwierdzono, że rozkłady a priori o lżejszych ogonach, takie jak rozkład Laplace'a, są w tym zastosowaniu bardziej skuteczne niż te o cięższych

ogonach. Opracowany system pętli decyzyjnej z udziałem człowieka (HITL) jest w stanie zautomatyzować blisko 40% zgłoszeń z precyzją na poziomie 98%, jednocześnie skutecznie przekazując próbki o niskim stopniu pewności do ręcznej analizy przez ekspertów. Chociaż wartość 40% może wydawać się niewielka w porównaniu z teoretycznymi wynikami często przytaczanymi w badaniach nad uczeniem maszynowym, w kontekście przemysłowym stanowi ona znaczący uzysk. Automatyzacja niemal połowy zgłoszeń, z których każde wymaga zwykle od kilku godzin do kilku dni analizy przyczyn źródłowych, przekłada się na znaczne oszczędności czasu i kosztów, umożliwiając programistom wydajniejsze rozwiązywanie problemów i szybsze dostarczanie poprawek.

Przedstawiając dwa odrębne, autorskie rozwiązania, niniejsza rozprawa dostarcza wzorzec dla rozwoju wydajnej i opłacalnej automatyzacji w kluczowych procesach inżynierii oprogramowania. Rozprawa ukazuje kluczową zmianę w podejściu – od dążenia wyłącznie do maksymalizacji dokładności na rzecz budowy niezawodnych, interpretowalnych oraz świadomych własnej niepewności systemów ML. Przyczynia się to do zwiększenia odporności i wydajności procesów rozwoju oprogramowania w nowoczesnych sieciach telekomunikacyjnych.

Słowa kluczowe: 5G, Bayesowskie sieci neuronowe (BNN), Klasyfikacja błędów, Pętla decyzyjna z udziałem człowieka (HITL), Inżynieria oprogramowania, Kwantyfikacja niepewności, LMLDA, Niepewność epistemiczna, Optymalizacja zestawów testowych (TSO), Telekomunikacja, Testowanie oprogramowania, Uczenie maszynowe, Wielkie modele językowe (LLM), Wykrywanie anomalii

Contents

List of Figures

List of Tables

List of Symbols

General notation

C	A class or category
$\text{dist}(\mathbf{x_i}, \mathbf{x_j})$	Generic distance between vectors $\mathbf{x_i}$ and $\mathbf{x_j}$ (≥ 0)
ϵ	Error term or a small positive constant (> 0)
K	Number of distinct categories (e.g., clusters) ($\in \mathbb{N}^+$)
m, n, N, M	Number of items (e.g., examples, cases, clusters, faults, tokens, matrix dimensions, documents in corpus (LDA)) ($\in \mathbb{N}^+$)
p	Probability value ($\in [0, 1]$)

Machine Learning Concepts

J	Loss or cost function (≥ 0)
θ	Model parameters (e.g., weights, biases); in LDA, topic distribution for a document ($\in [0, 1]^K$, sums to 1)
w	Weight of a model parameter ($\in \theta$)
$\mathbf{X}$	Input data matrix or dataset (e.g., textual documents)
$\mathbf{x}$	Input vector or data point ($\in \mathbf{X}$)
$\mathbf{Y}$	Set or matrix of true or ground truth values
$\mathbf{y}$	Target value or true label ($\in \mathbf{Y}$, e.g., $\in \{0, 1\}$ for binary classification, $\in \mathbb{R}$ for regression)
$\hat{y}$	Predicted value (e.g., $\in [0, 1]$ for probability, $\in \mathbb{R}$ for regression)

$\mathbf{F}$	Set or matrix of specific data features (attributes used as input to a model)
f_i	The i-th specific feature, typically a component of an input vector $\mathbf{x}$ ($\in \mathbf{F}$)
σ	Sigmoid (logistic) activation function
softmax	Softmax activation function (a generalization of sigmoid for multi-label problems)

Software Delta & Testing

$\mathbf{D}$	Dataset of software deltas (matrix)
$\mathbf{d}$	Software delta: a specific instance of $\mathbf{x}$ ($\in \mathbf{D}$)
$\mathbf{T}$	Set of tests
t	A single test ($\in T$)
τ	Execution duration of a test (or general time)
$\mathbf{C}_s$	Set of software components ($\{c_1, \ldots, c_{250}\}$)
$\mathbf{F}_t$	Set of supported file types ($\{\text{cpp}, \ldots, \text{other}\}$)
Δ_c	Set of change types ($\{\text{additions}, \text{deletions}\}$)

Uncertainty Quantification

$\omega_a^2(\mathbf{x})$	Aleatoric uncertainty for input $\mathbf{x}$ (≥ 0)
$\omega_e^2(\mathbf{x})$	Epistemic uncertainty for input $\mathbf{x}$ (≥ 0)
$\omega_{\text{total}}^2(\mathbf{x})$	Total uncertainty for input $\mathbf{x}$ (≥ 0)

Clustering Metrics

$DBI(i)$	Davies-Bouldin Index for point i (≥ 0)
$s(i)$	Silhouette coefficient for point i ($\in [-1, 1]$)
$a(i)$	Average intra-cluster distance for point i (≥ 0)
$b(i)$	Minimum average inter-cluster distance from point i (≥ 0)

Sequence & Language Modeling

S	A sequence or set of tokens
s_n	The n-th token in a sequence
ℓ_{CE}	Cross-entropy loss (≥ 0)

$\mathbf{v}_w$	Vector embedding for a word or phrase
$\mathbf{v}_{g_i}$	Vector embedding for the i-th n-gram

Attention Mechanism

$\mathbf{Q}$	Query matrix
$\mathbf{K}$	Key matrix; in LDA, total number of topics ($\in \mathbb{N}^+$)
$\mathbf{V}$	Value matrix
d_k	Scaling factor (typically dimension of key vectors) ($\in \mathbb{N}^+$)

Probabilistic Models & Inference

$p(e_n)$	Probability of an event e_n ($\in [0, 1]$)
$q_\theta(w)$	Approximate posterior distribution over weights w, parameterized by θ
$p(w\|\mathbf{D})$	Posterior distribution of weights w given data $\mathbf{D}$
$p(\mathbf{D}\|w)$	Likelihood function of data $\mathbf{D}$ given weights w
$p(\mathbf{Y}\|\mathbf{X}, w)$	Likelihood function of labels $\mathbf{Y}$ given inputs $\mathbf{X}$ and weights w
$p(w)$	Prior distribution on the weight w
$p(\mathbf{D})$	Evidence of the model (marginal likelihood)
$\mathcal{L}(\theta)$	Evidence Lower Bound (ELBO)
$\mathbb{E}_{q_\theta(w)}[\cdot]$	Expectation with respect to $q_\theta(w)$
$\mathrm{KLD}(q_\theta(w)\|\|p(w))$	Kullback-Leibler Divergence (≥ 0)
$\mu(\mathbf{x}, w)$	Mean prediction of the network for input $\mathbf{x}$ with weights w
$\mathrm{Var}_{q(w)}[\cdot]$	Variance with respect to $q(w)$ (≥ 0)
$\boldsymbol{W}$	Observed words in the corpus (LDA)
$\boldsymbol{Z}$	Latent topic assignments for words (LDA)
$\boldsymbol{\varphi}$	Word distribution for each topic ($\in [0, 1]^V$, sums to 1) (LDA)
α	Dirichlet prior parameter for document-topic distributions (> 0) (LDA)
β	Dirichlet prior parameter for topic-word distributions (> 0) (LDA)

List of Acronyms

3GPP 3rd Generation Partnership Project

APFD Average Percentage of Fault Detected

APR Automated Program Repair

BCE Binary Cross-Entropy

BNN Bayesian Neural Network

BPE Byte Pair Encoding

CCE Categorical Cross-Entropy

CD Continuous Deployment

CI Continuous Integration

CNN Convolution Neural Network

DBI Davies-Bouldin Index

ECE Expected Calibration Error

ELBO Evidence Lower Bound

FN False Negatives

FP False Positives

GLU Gated Linear Unit

GNN Graph Neural Networks

GPT Generative Pre-trained Transformer

GQA Grouped-Query Attention

HAC Hierarchical Agglomerative Clustering

HITL Human-In-The-Loop

KLD Kullback-Leibler Divergence

LDA Latent Dirichlet Allocation

LLM Large Language Model

LMLDA Linear Model of LDA

LoRA Low-Rank Adaptation

LSTM Long-Shor Term Memory

LTE Long Term Evolution

MCD Monte Carlo Dropout

MHA Multi-Head Attention

ML Machine Learning

MQA Multi-Query Attention

NLP Natural Language Processing

NN Neural Networks

OOD Out of Distribution

OOV Out Of Vocabulary

PEFT Parameter-Efficient Fine-Tuning

PPL Perplexity

RAG Retrieval-Augmented Generation

RAN Radio Area Network

RNN Recurrent Neural Network

SBTS Single RAN Base Station

SFT Supervised Fine-Tuning

SVI Stochastic Variational Inference

SVM Support Vector Machine

TN True Negatives

TP True Positives

TPR True Positives Rate

TSO Test Set Optimization

TSP Test Set Prioritization

TSR Test Set Reduction

UE User Equipment

VI Variational Inference

1 Introduction

The ongoing evolution of telecommunication systems, marked by the complex transition from 4G to 5G and beyond, presents significant challenges to software development. This trend reflects the profound software-defined methodology of the telecommunications industry, where functions traditionally handled by specialized hardware are now implemented as complex software. Consequently, this work approaches these challenges from a modern computer science perspective, applying software engineering methods to the network management plane. This approach is different from the methods traditionally employed in telecommunications engineering, which typically focus on hardware and signal processing.

Modern mobile network base stations are among the most sophisticated and critical systems built today. An **anomaly** in its behavior, specifically a **software defect** – that is, a bug introduced by a programmer, not a random operational fault – can trigger a chain of failures, causing a wide societal disruption. Millions of people could experience call drops and service outages on a busy morning, disrupting emergency services, financial transactions, and businesses. Critical hospital devices could be disconnected and public transport could be halted. These are not just theoretical risks; in November 2024, Denmark faced a severe network breakdown due to a routine software update error, which affected the TDC mobile network [119]. This forced hospitals to scale back non-urgent services and required security personnel to aid those unable to seek help. This example

highlights the urgent need for robust software quality processes to quickly detect and address code anomalies before deployment.

This dissertation is the result of an **industrial doctorate**, a research and development initiative of **AGH University** and **NOKIA**. It addresses the challenges of automation and software testing within the CI/CD pipelines of these complex telecommunication systems. In this context, NOKIA's Single RAN Base Station (SBTS) serves as a crucial technological foundation, integrating 2G, 3G, 4G and 5G into one complex platform. Then, for such a product, the core problem lies in the growing inefficiency of traditional quality assurance methods in the face of continuous software evolution. Key challenges include:

- **Test suite decay**: As software evolves, large suites of static regression tests become progressively less effective, time-consuming, and resource-intensive, leading to increased technical debt. At some stage, no one in the testing team recalls the aim of certain tests and rarely controls if they are still relevant to any current software function. With repositories containing hundreds of thousands of tests, manual verification becomes nearly impossible.

- **The automation dilemma for the use of machine learning (ML) systems**: Although large language models (LLM) seem promising for processing large amounts of textual data (e.g. automated test case, bug reports, logs), their adoption introduces significant risks. These include *false savings* from outsourced infrastructure costs that are offset by expensive API maintenance, a *harmful interdependence* on external models whose changes can break internal processes and concerns about data privacy. Furthermore, an overreliance on prompting external models risks degrading the core software engineering expertise within an organization.

- **The need for reliable automated systems**: In a high-stakes environment, automation cannot be a "black box". To be adopted by engineers, the system augmented with machine learning (ML) applications must be reliable and aware of its own limitations. The aim is not to replace human experts, but to increase their productivity by using a reliable human-in-the-loop (HITL)

system that boosts process trust and engineer satisfaction, enabling them to focus on more crucial tasks.

This research rejects the dependence on external ML systems or models and instead advocates the development of **proprietary applications** specialized to the specific needs of the telecommunications domain. Crucially, this work clarifies that **detection** in base stations refers to the process of identifying software defects during the pre-deployment testing and quality assurance phase. The focus is firmly on the software engineering lifecycle, not on real-time monitoring of live operational networks. Then, the **uncertainty-aware** ML systems are designed to predict outcomes and assess their confidence, allowing low-confidence cases to be referred to human engineers, thus improving trust in automation. By bridging academic theory with corporate implementation within this specific context, this work demonstrates how to build practical, resource-efficient, and trustworthy solutions to confront complex industrial challenges.

1.1 Thesis statement and contributions

The central thesis of this dissertation is formulated as follows:

> **The automation of anomaly detection in the software of cellular network base stations can be made significantly more efficient and reliable by integrating proprietary, uncertainty-aware ML frameworks directly into the pre-deployment CI/CD testing pipeline.**

The core of this work involves the development of such a framework, realized as an anomaly management pipeline designed to automatically identify software anomalies, detect duplicates of existing bug reports, and classify new findings to the appropriate engineering team. By design, the system escalates ambiguous cases for manual intervention. These contributions demonstrate that such frameworks can significantly improve the efficiency of test optimization and the reliability of bug management, moving beyond simple automation to create effective, human-centric systems.

To validate this thesis, this dissertation presents two major complementary contributions.

1. **A framework for resource-efficient test set optimization (TSO)**. This contribution addresses the problem of selecting the most effective subset of regression tests to detect new anomalies. It culminates in the development of the linear model of **latent Dirichlet allocation (LMLDA)**, a model that prioritizes tests by analyzing the relationships between code changes and test descriptions. This provides an efficient and internal alternative to resource-heavy LLM models.

2. **A framework for reliable, uncertainty-aware bug triage**. This contribution confronts the automated classification of incoming software bug reports found by the LMLDA, a task where managing the uncertainty of prediction is as important as accuracy. It introduces a highly specialized **two-stage Bayesian LLM framework** designed to quantify epistemic uncertainty. To enrich the context of bug reports, the framework first employs a context-aware pre-tokenization method that automatically expands domain-specific acronyms and abbreviations. This ensures a richer and more comprehensive input for the model. As a second step, a Siamese neural network is used to identify anomalies that have already been discovered. This is particularly crucial for a worldwide company as it helps reduce the number of false positives in the dataset and saves significant engineering effort by preventing multiple teams from trying to determine the root cause of the same issue in parallel. The system then learns to identify the cases it is uncertain about, allowing it to reliably automate the majority of routine tickets while escalating complex or ambiguous ones to expert engineers. This approach promotes a safe and efficient HITL workflow.

Together, these contributions provide a comprehensive methodology for building and deploying automated software detection pipelines that are specialized for the demanding requirements of critical systems, ensuring data control, process reliability, and engineering trust.

1.2 Structure of the dissertation

This dissertation is structured to progressively build a cohesive understanding of the problems and solutions, from foundational concepts to practical, validated implementations.

- **Chapter 2**: This chapter sets the foundations for the dissertation by covering the core concepts and the data landscape. It provides a comprehensive overview of the software development lifecycle and the anomaly detection pipeline, highlighting the distinct data types in software anomalies: binary, numerical, categorical, and textual. The chapter then briefly explores machine learning methodologies, from classical supervised and unsupervised models to advanced neural networks, focusing on LLMs and their uses. Additionally, it explores probabilistic methods, such as the Bayesian Theorem and Bayesian Neural Network (BNN), to set up the uncertainty-aware frameworks discussed later. This foundation is essential for understanding the challenges and solutions addressed in the subsequent chapters.

- **Chapter 3**: This chapter presents a comprehensive literature review, summarizing the current state of the art in machine learning techniques applied to software testing and providing a theoretical foundation for the contributions of this dissertation. It discusses unsupervised techniques such as clustering and topic modeling, along with supervised and deep learning strategies, including traditional and neural network methods. In addition, it examines probabilistic strategies for uncertainty estimation, discussing various sources of uncertainty inherent in software engineering tasks.

- **Chapter 4**: This chapter presents the first original contribution. It details the development of the TSO framework, starting with clustering techniques to identify high-potential test subgroups and culminating in the proposal and validation of the **LMLDA** model.

- **Chapter 5**: This chapter presents the second major contribution: a deep dive into the use of LLMs in a real-world industrial scenario. The chapter opens with the challenges of using standard LLMs in software engineering. It introduces a focused pre-tokenization method that automatically

expands domain-specific acronyms, enriching the model's input context. In addition, it outlines the use of a Siamese neural network to detect duplicate bug reports, crucial to reduce false positives and conserving engineering efforts across globally distributed teams. Building on the foundations laid in previous chapters, this chapter further details the design, implementation and extensive evaluation of a **two-stage Bayesian LLM framework** for anomaly classification.

- **Chapter 6**: The final chapter synthesizes the research findings, discusses their broader implications for the telecommunications industry and software engineering practices, and outlines promising directions for future research.

2 Theoretical foundations

This chapter establishes the theoretical foundation for novel anomaly detection methods developed and implemented in an industrial setting. The research originates from real-world challenges at NOKIA. This industrial background ensures that both the theoretical framework and practical implementation address the actual challenges in the development of large-scale telecommunications software.

The subsequent sections present a comprehensive framework for anomaly detection in software systems. Section 2.2 introduces the anomaly detection pipeline, establishing the definition of the problem, mathematical foundations and performance metrics tailored to industrial needs. This pipeline, built on telecommunication datasets, optimizes regression testing and bug classification while incorporating uncertainty quantification. The data landscape (Section 2.3) demonstrates the complexity of inputs through binary, categorical, numerical, and textual examples from production systems. Building on this foundation, Section 2.4 examines traditional machine learning approaches as baseline methods, while Section 2.5 explores advanced deep learning architectures, particularly focusing on adaptation of LLM to specialized telecommunication domains. The framework culminates in Section 2.6 with probabilistic approaches, integrating deterministic and generative modeling through LDA, BNN, and Variational Inference (VI), ultimately enabling robust uncertainty quantification that allows models to effectively recognize their knowledge boundaries in practical applications.

2.1 The software life-cycle

Automated testing provides historical data that can be used to predict potential problem areas, and optimize the test suite by selecting and prioritizing test cases. By automating repetitive and time-consuming tasks, developers can focus on more critical aspects of the project, such as designing new features and improving the overall architecture of the software. This not only improves the accuracy and efficiency of the testing process, but also reduces the overall workload, allowing developers to focus on more critical tasks. In addition, automated testing can provide faster feedback, allowing teams to identify and address issues early in the development cycle [109].

A defined framework, CI and CD commonly referred to as **CI/CD** and demonstrated in Figure 2.1, is essential and foundational in modern software development. It provides a structured methodology for identifying, analyzing and addressing potential issues in software. It functions as a standardized methodology that directs the testing procedure and ensures uniformity between various projects and teams. At the core of CI/CD is the automation of code integration from multiple developers into a shared repository. In this repository, the code undergoes automated tests and builds in a continuous feedback loop to identify potential defects early in the development lifecycle.

The **CI phase** automates the integration of code contributions from various developers into a common repository, initiating an automatic build and testing sequence. Subsequently, a series of automated tests is performed, including unit, integration, and regression tests, to confirm accuracy and ensure correctness and stability of the code. The subsequent steps integral to the entire phase can be outlined as follows:

- **Planning** involves defining the scope, objectives, and requirements of the project. It ensures that all stakeholders understand the development goals and establishes a roadmap.
- **Coding** is the implementation phase in which developers write source code based on planned requirements. Using IDEs and version control systems such as Git, teams collaborate to create software while maintaining code synchronization.

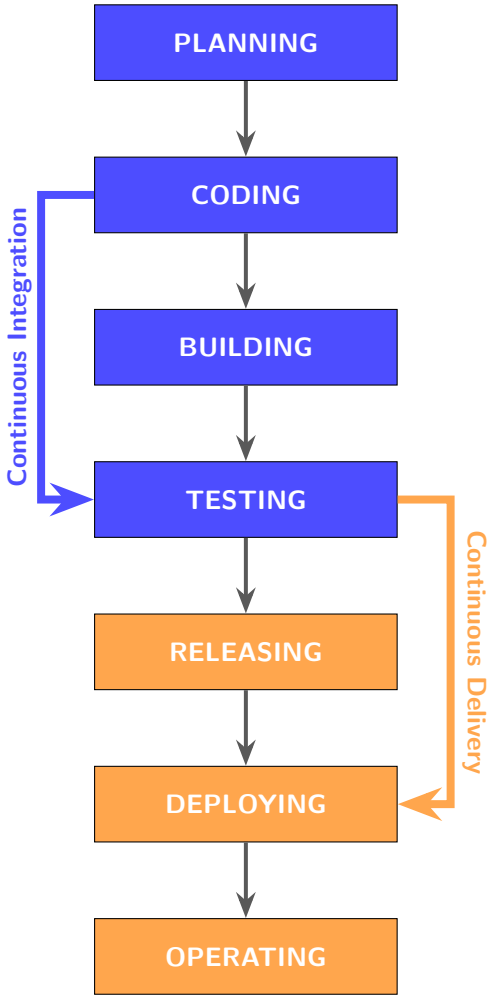


Figure 2.1: An overview of the CI/CD pipeline. The automated processes of code integration, development, and deployment is highlighted. A thorough testing phase, which includes procedures such as regression testing, receives particular attention

- **Building** transforms the source code into executable files through compilation.
- **Testing** verifies code functionality through automated and manual processes. Automated testing frameworks run various tests when new code is committed, while testers perform manual verification to ensure software quality and reliability.

Once the code has successfully passed the CI stage, it moves into the **CD phase**, which automates the release of the code to production or staging environments. It eliminates manual release efforts and ensure quick and error-reduced deployment of new features or bug fixes to users. This procedure typically covers a variety of deployment strategies, including rolling updates and normal releases. The steps involved in CD can be outlined as follows:

- **Testing** in CD occurs in a staging environment that mirrors the production conditions. This phase verifies the behavior, performance, and stability of the application in realistic scenarios before the release of the customer.
- **Releasing** prepares the application for deployment by documenting changes and ensuring all pre-deployment requirements are met to maintain deployment integrity.
- **Deploying** makes the application available to end users through automated processes.
- **Operating** involves monitoring and maintaining the application in the production environment.

With the acceleration of software development for products such as a 5G base station, more and more special emphasis is placed on anomaly detection in the CI part, which is a loop that connects **software development (code) and testing**, as shown in Figure 2.1. This emphasis is on purpose; automated testing is integrated into both the CI and CD pipelines to provide immediate feedback on code changes and to identify problems early. This closed loop of continuous testing creates a solid framework where testing is not just a phase, but an integral part of the entire

CI/CD pipeline. Hence, the **software bug** (or **anomaly detection**) performed through the testing is vital to maintain the reliability of the software system, targeting problems such as failures and vulnerabilities; it does not produce new bugs, but simply discovers their existence [89].

In this context, NOKIA's **Single RAN Base Station (SBTS)** serve as a crucial technological foundation, integrating 2G, 3G, 4G and 5G into one complex platform. This convergence offers a substantial opportunity for industrial research on software reliability and anomaly detection. This is particularly significant given the increasing complexity of telecom software and the critical function of the CI/CD pipeline, both of which require advanced anomaly detection techniques for reliable systems. Consequently, this dissertation analyzes data from these modern base stations to effectively represent actual network behaviors, and, furthermore, the SBTS platform is the testing ground for the automated anomaly detection methods designed and developed in the course of this PhD dissertation.

2.2 The anomaly detection pipeline

Software testing, at its core, is a data-driven process focused on uncovering software anomalies, systematically exploring potential issues based on predefined inputs and expected behaviors, rather than relying on chance or arbitrary checks. Figure 2.2 illustrates the key data sources and their relationships in the testing framework: software changes trigger tests that can uncover anomalies.

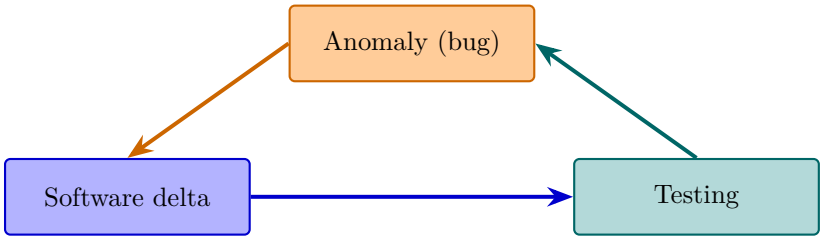


Figure 2.2: The relationship between data parts that creates a whole dataset: software modifications (delta) necessitate testing, which may reveal anomalies (bugs)

In its essence, anomaly detection seeks to identify patterns or behaviors that notably differ from the norm, utilizing clear data structures and their interactions. In telecom, this may involve minor network traffic changes, unforeseen equipment problems, or covert intrusions and service dips. However, in such complex settings, achieving reliable anomaly detection is challenging.

The section is structured as follows. The initial focus shifts to the optimization of regression testing, introducing the optimization framework and its role in efficiently identifying critical issues in complex test suites. The subsequent discussion then covers classification and analysis of anomalies, exploring the essential process of classifying identified issues.

2.2.1 The problem overview

The increasing complexity of telecommunication software requires sophisticated approaches to anomaly detection and resolution. Figure 2.3 broadly presents a CI/CD pipeline, highlighting key areas for automated anomaly detection and optimization. This three-part approach focuses on optimizing regression test sets

to using machine learning to classify and resolve bugs faster, and implementing confidence estimation to ensure system trustworthiness. This balanced framework not only promises improved efficiency and reliability in complex telecom environments, but also acts as a blueprint for automated and optimized anomaly management in the research effort carried out and outlined in subsequent chapters.

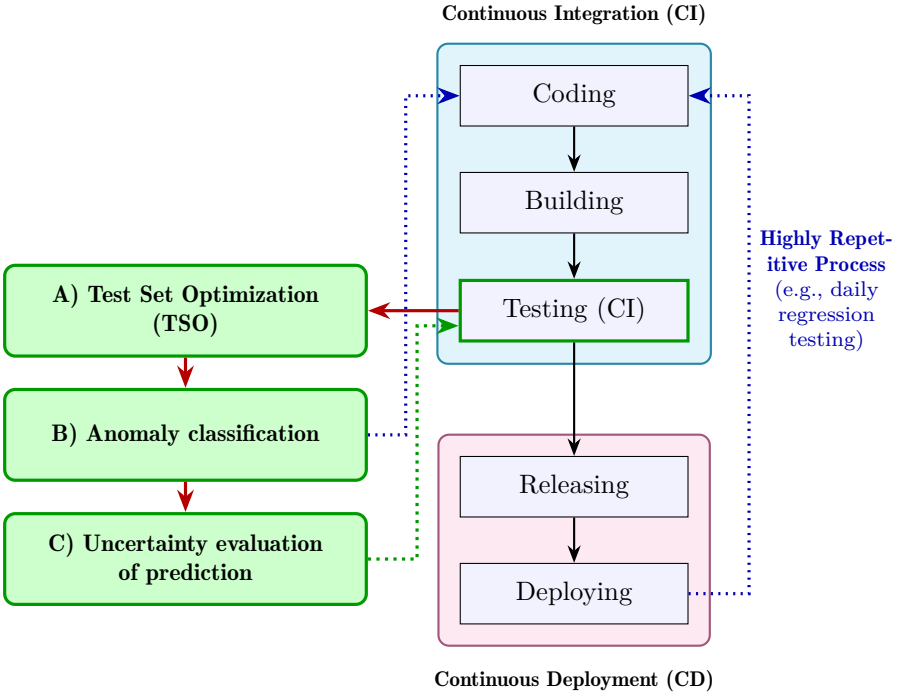


Figure 2.3: The general CI/CD pipeline diagram with proposed improvements related to automated anomaly detection

A) The TSO in regression testing

This step involves re-executing tests on modified software to confirm that recent changes have not negatively impacted existing features. It is a continuous process, but the execution of a whole testing set can be performed over different time

intervals, adapting to the needs of the project and the development pace. Daily or periodic tests can be distinguished, each serving different purposes in the quality assurance framework. Periodical tests are usually run multiple times during a single software cycle, providing deeper insights into system behavior and potential issues at critical development stages. For simplicity, let us refer to these cycles by calling them *sprints*, following the terminology of Agile methodology [12, 116]. In this context, sprints are time-boxed events of typically two weeks duration, in which the development team focuses exclusively on delivering sprint goals, including comprehensive testing activities.

Unlike traditional testing methodologies, this approach orchestrates a complex, synchronized testing ecosystem where multiple components work in harmony. The process is started within a fixed timeframe, such as the first hours of each first day of a sprint, allowing for a daily overview of system stability and quick detection and resolution of new issues. The tests are performed as coordinated sets rather than isolated units, and the results are aggregated and analyzed to provide comprehensive coverage. This collaborative framework involves multiple testing teams working in parallel, each focusing on specific components or functionalities, but synchronized within the same software development cycle. Rather than being a rigid monolithic process, regression testing provides an adaptable testing schedule within the specific time constraint, allowing for varied testing speeds for different components while maintaining coordinated effort across distributed teams within a sprint. Although the number of tests performed steadily decreases with time, Figure 2.4 reveals that the distribution of bugs found is not uniform. Specifically in the dataset from a NOKIA process, a significant portion of bugs are discovered during the third and sixth days of testing, suggesting a particular pattern in bug discovery here. Initially queued on the sprint's first day, some tests face delayed execution until the testing queue shortens, enabling subsequent tests to proceed.

Each software iteration undergoes extensive regression testing, ranging from several hundred to more than a thousand tests, depending on the hardware and product complexity. Every **test** is regarded an abstract entity with two potential outcomes: **failed** or **passed**, with a lack of results implying a pass. By comparing two consecutive software versions, we assess their **delta** – the differences between

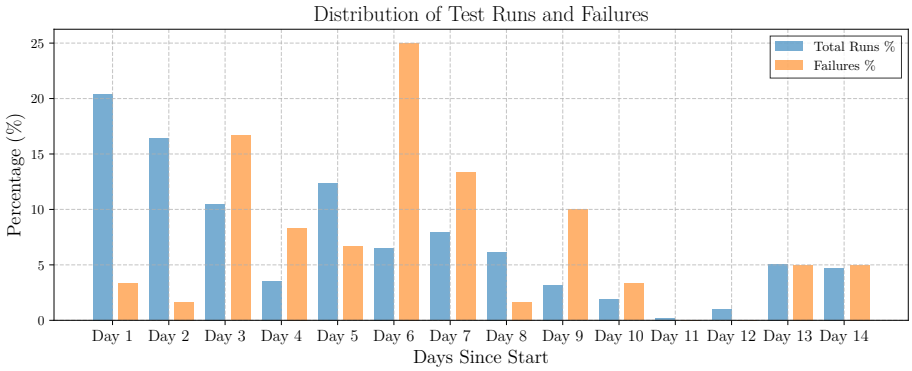


Figure 2.4: Based on data from NOKIA's periodic regression testing cycle for its Single RAN product, averaged over several months and sprints, the figure illustrates that the distribution of test executions and bugs in this specific scenario is not uniform, reflecting patterns unique to this technological and testing process

them. With change captured by version control systems such as Git or SVN, we can precisely quantify lines of code added or removed across components. This enables us to study the evolution of the software and correlate changes with test results. Therefore, the regression testing process, as shown in Figure 2.5, depends on two interconnected factors. The first, the software build (or testing **candidate**, highlighted in **blue**), represents code changes, formally expressed as $\mathbf{D} = \{d_1, d_2, \ldots, d_n\}$, where D is an individual code delta and $n = |\mathbf{D}|$ is the total number of modifications. We regard a delta as a group of file changes within a specified period, such as a day. There is an assumption that at least one new candidate is produced daily through the CI/CD pipeline, generating a sequence of deltas D_n, where $n \in \mathbb{Z}$ represents the number of deltas to which the test suite is applied. The second factor, the regression **testing suite** $\mathbf{T} = t_1, t_2, ..., t_m$ (marked in **orange**), comprises a collection of tests performed in each testing cycle. Each delta is related to specific version of testing suite to which it was applied. Hence, the **regression suite T** serves as the primary source of test execution data, captured in the known results matrix $\mathbf{Y} = [y_{ij}]$, where $i \in 1, 2, ..., m$ represents the test index and $j \in 1, 2, ..., n$ represents the delta index. Thus, y_{ij} indicates whether the i-th test passed (0) or failed (1) on the j-th delta.

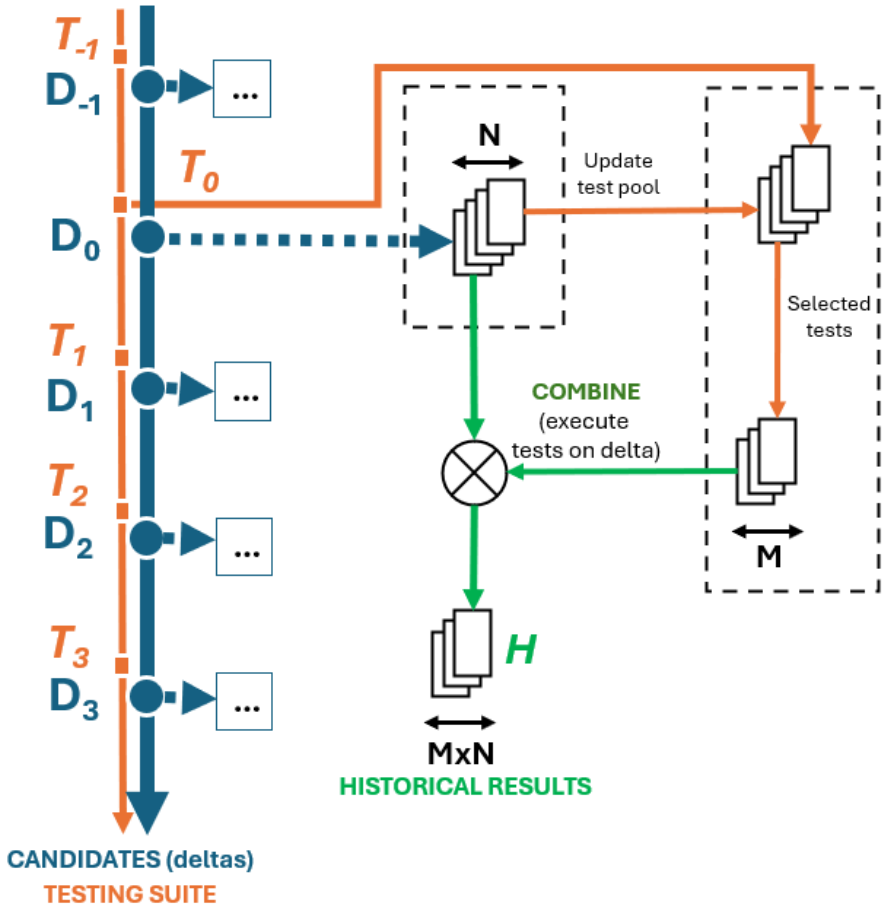


Figure 2.5: Illustration of the regression testing procedure showing the mapping between software deltas and test suites, with the interaction point highlighted in green. The process produces a binary results matrix indicating pass / fail outcomes for each test-delta combination

At some point, it might be beneficial to reassess the approach and analyze whether certain commits in the CI can be omitted [2]. These files contribute to certain functionalities and can be considered as integral components of larger software modules. As the entire codebase is compiled into a single cohesive software product, each test plays a critical role in assessing the stability of individual files. The intersection between software builds (delta $\mathbf{D}$) and the testing suite $\mathbf{T}$ creates a space where we can observe up to $m \times n$ potential interactions. Each interaction point (t_i, d_j), emphasized in **green**, represents a potential location for **software anomalies** (commonly referred to as **bugs**), with known results $y_{ij} \in Y$ indicating previous test results (pass / fail). The optimization problem becomes finding an optimal subset of $\mathbf{T}$ – specifically, one that maximizes fault detection while minimizing total execution time $\sum_{i=1}^{m} \tau_i$, while considering test priorities and coverage requirements defined by testing organization processes.

The key challenge in regression testing is **the gradual degradation of the identity of tests** over time. As software systems continue to grow in complexity, new features and fixes are continuously integrated. The number of tests increases over time. The growing test suite consumes resources and obstructs the overall development process, highlighting the need for efficient test management. Furthermore, the test suite risks gradual degradation over time. As knowledge drifts and developers and testers move to different tasks, the purpose of some tests in the suite becomes unclear. Initially designed with care to assess critical functions, these tests gradually lose their distinctiveness. The introduction of new tests further obscures the original intent of certain cases. The problem is worsened by the fact that sometimes certain tests do not find errors throughout a long period of time or even at all during their entire lifespan. In NOKIA's terminology, these are termed "evergreen", with red tests indicating failures and green representing error-free runs. The emergence of other unreliable tests, termed "shaky" or "flaky", significantly challenges efficient testing. Shaky tests fail sporadically due to factors unrelated to code changes, whereas flaky tests are essentially unstable, failing consistently due to environmental nuances or design issues. These unreliable tests compromise the integrity of the CI, complicating the differentiation between actual software bugs and false positives. Consequently, developers and testers face substantial challenges in discriminating between tests that fail due

to correct software functionality versus those that fail due to misalignment with their specified requirements.

Additionally, in the case of such a sophisticated system as Single Radio Area Network (RAN) base stations, automatic tests are also described with additional information, such as the technology area they cover or the exact hardware versions they are intended for. For example, a bug that has been detected for a 3G technology can temporarily block the entire set of tests intended for that technology. Depending on the timing of the event, the bug can affect up to a dozen percent of the test queue. In this case, if the system detects even one error, all tests in a given area are immediately blocked until the problem is fixed. As a result, it can be said that from the perspective of historical data, **software bugs in regression are more wide-ranging than longer-lasting**. This means that one bug affects many tests (it expands on the whole testing area), although the time to fix it varies.

As the software evolves, the core of the test suite remains static; yet new test cases are introduced. The suite is rarely reduced, aligning with the belief that excessive testing is preferable to missing any bugs. Over time, ever-growing and unreliable test results disrupt the testing queue, creating a dangerous situation. No one risks removing a test from the suite due to the potential risk of unintentionally allowing undetected bugs to pass through the pipeline. It is challenging to calculate the cost associated with the software bug. However, in 2012, Google staff estimated that the average expense of correcting a software bug was $1500, with almost 40% of the company's employees participating in the bug correction process [131]. Considering this impact of testing cost – especially given time and resource limitations – optimizing the regression testing suite is crucial, even if these constraints apply to general testing and not just regression.

Automation addresses these challenges by enabling frequent and well-structured testing cycles with reduced manual effort. However, the sheer volume of tests and the need for rapid feedback demand more sophisticated approaches. The **cloud-based testing** emerges as a solution [62], providing scalability that supports CI/CD practices for faster and more reliable software releases.

Now, there is a common scenario where a cloud testing infrastructure runs at maximum potential but is constrained by slow data processing due to the large

volume of test results. Selecting representative test lines that can verify everything from basic connectivity to high-throughput performance requires careful analysis. This is where TSO and Test Set Prioritization (TSP) as its specific form, become crucial. TSO minimizes test suite size while maintaining fault detection capability by focusing on essential, non-redundant tests. TSP arranges tests to prioritize critical fault discovery, which is particularly valuable when resources or time are limited.

B) Bugs classification

When test failures occur, an intelligent **bug classification system** becomes essential in the anomaly detection pipeline. This system, integrated after TSO, employs sophisticated ML algorithms to categorize failures and accelerate root cause analysis. Building upon these insights, particularly the identified software anomalies, we define the smallest unique piece of information related to these anomalies or incidents as a *ticket*. Although a ticket in this dissertation primarily represents a description of an observed software problem (bug), it is a universally applicable concept used in issue tracking across various domains. Each ticket provides a textual description of an observed software problem, which is then forwarded to a designated software development department responsible for preparing a source code fix. If a ticket represents a piece of data, its assigned department serves as its label. Consequently, an automated pipeline for training and classification must be designed.

However, several significant challenges emerge in the implementation of such a system. Paradoxically, as overal quality improves and bugs are fixed more efficiently, we face an **inverted data problem** – the very success of the system leads to fewer examples for training. Additionally, the ML models struggle to understand complex, unstructured data formats such as system logs and human observations, which often contain crucial contextual information. The problem is further complicated by the massive data imbalance between different departmental categories and frequent context shifts driven by evolving technology and changing market demands.

Occhipinti et al. [90] offer a general overview of recent approaches in text classification pipelines. Drawing from these observations, we propose our own

pipeline. In this context, a ticket is represented as a vector composed of S words or sub-word parts (tokens) and is assigned to a class c.

To formalize this, the core problem is to determine the **conditional probability** $p(c, \mathbf{v}) = p(c|s_1, ..., s_n)$, where $\mathbf{v}$ denotes the vector of tokens $(s_1, ..., s_n)$. Here, $n \in \{0, 1, ..., N\}$ represents the number of tokens in a ticket and $c \in C$ denotes the assigned department. In such circumstances, a text sample describing a software bug acts as an observation (or evidence) associating a set of tokens with a defined class.

Therefore, the objective is to find a posterior probability distribution indicating the likelihood that a new sample will be associated with one of these departmental labels. The application of **Bayesian reasoning** is a natural implication, leading to the use of BNN (ideas explained in a Section 2.6), as Blundell et al. [17] demonstrated that such an architecture possesses the capability for uncertainty estimation.

C) Addressing uncertainty

As shown in Figure 2.6, the automated pipeline must address various types of uncertainty while maintaining user interaction points. Anomalies, sometimes called incidents, are defined by the internal corpus that describes all the exclusive context related to the area of research. The application of **Bayesian reasoning** through BNN provides the ability to learn the in-context representation in the form of embedding matrix and then in the inference phase to estimate uncertainty. This framework addresses both aleatoric uncertainty and epistemic uncertainty, which is crucial when dealing with domain-specific terminology and rapidly evolving technology landscapes.

These uncertainty types significantly impact the entire pipeline. In TSO, aleatoric uncertainty manifests itself in the inherent variability of test outcomes, while epistemic uncertainty reflects our limited knowledge about the system's behavior under new conditions. During classification, aleatoric uncertainty captures the ambiguity in ticket descriptions and natural language variations, while epistemic uncertainty represents the model's knowledge gaps about new technological contexts or rare failure modes.

Most critically, these uncertainties directly influence human trust in the au-

tomated system. Engineers, accustomed to deterministic debugging processes, may be skeptical of probabilistic classifications. However, by explicitly quantifying and communicating both types of uncertainty, we enable engineers to make informed decisions about when to trust the system's recommendations and when to apply their expertise. This transparency in uncertainty estimation becomes the bridge between automated efficiency and human expertise, crucial for maintaining system reliability while advancing towards greater automation in telecommunications testing and debugging processes.

The integration of this classification system with TSO creates a continuous improvement cycle: once fixes are implemented, the optimized test suite can be executed again. This approach represents a paradigm shift from brute-force test execution towards an intelligent, targeted methodology. Although manual testing can be optimized, transitioning to cloud-based testing is essential for implementing these machine learning techniques and promoting full-scale end-to-end automation.

However, incorporating **ML techniques** into the CI testing phase poses significant challenges, mainly because the testing process, organization and telecommunication technology undergo frequent changes. This challenge of maintaining large volumes of high-quality data can be mitigated through careful uncertainty quantification and continuous model validation, ensuring the robustness of the automated anomaly detection pipeline while maintaining human oversight in critical scenarios.

2.2.2 Performance evaluation

In relation to Figure 2.5 (page 16), we see that the optimization problem is described by three matrices of known and constant shapes. The first **T** describes features of tests, such as their coverage, number of faults discovered, or more complex like frequencies of specific words used in a test. The second **D**, related to software deltas, is the numerical matrix that describes changes per component (columns) in each consecutive software version (rows). The third **Y** is a combination of values defining the outcome of each test. Tests that are not failed are usually marked with the boolean value False/0, while discovered failures are marked True/1. Therefore, the conclusion is that the TSO problem is of the

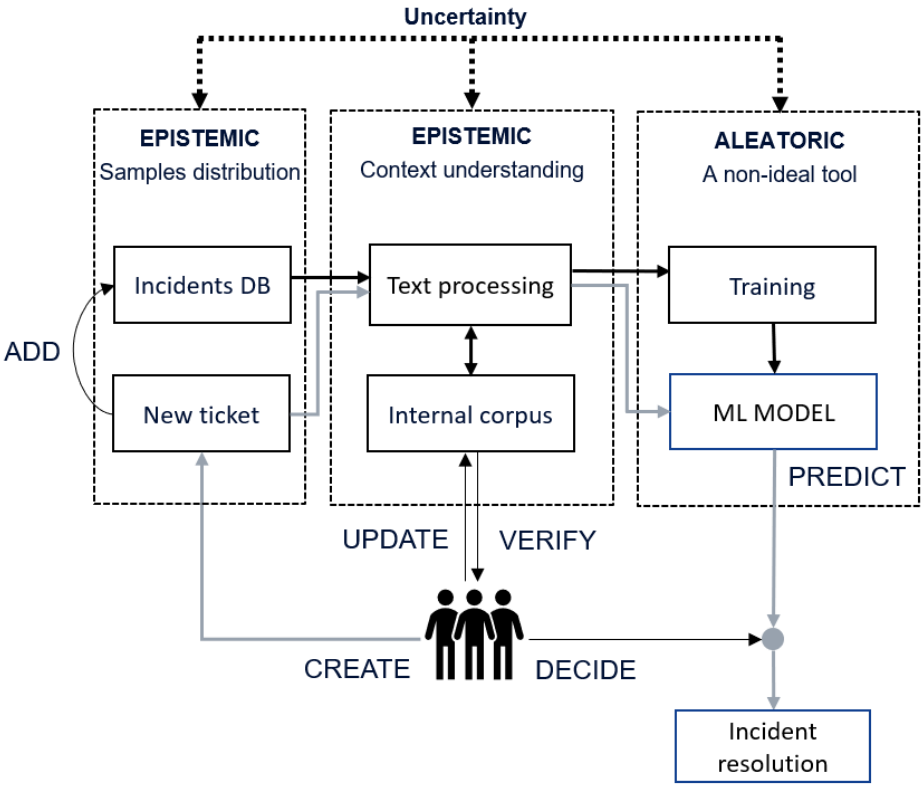


Figure 2.6: Typical anomaly (or incident) reporting and bug fixing process at NOKIA. Dark bold lines denote the typical machine learning pipeline process, while gray bold lines present the way a user interacts with it. Each step is connected with different limitations and uncertainty types, as well as user involvement with them

type of **binary classification**, in which the contents of the test and the software delta matrix are considered training data and the results of the tests describe the answers (labels or classes).

For a binary classification problem, we evaluate the performance of a model by analyzing four key outcomes derived from the confusion matrix: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). These values correspond to the number of true positives, true negatives, false positives (Type I errors), and false negatives (Type II errors), respectively. We can use these metrics to compute various performance measures, such as accuracy, precision, and recall.

Two critical metrics, Test Set Reduction (TSR) and Average Percentage of Fault Detected (APFD), provide crucial business insights:

- **Test Suite Reduction** measures the proportion of test cases removed from the original test suite. A higher value of TSR indicates a greater reduction in the number of test cases, indicating a more effective optimization strategy. The goal is to maximize it while minimizing any negative impact on the effectiveness of fault detection. It is calculated as:

$$TSR = 1 \; - \; \frac{TP + FP}{P + N} \tag{2.1}$$

- **Average Percentage of Faults Detected or** *APFD* [38] measures how effectively test prioritization speeds up fault detection. Essentially, a high *APFD*) value indicates that a prioritization technique reveals faults early, enabling quicker fixes. *APFD* is the weighted average of the percentage of faults detected as tests run in order and favors rapid fault detection. It ignores the total number of faults, relying on the assumption that all will be caught by the complete test suite.

$$APFD = 1 \; - \; \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2m} \tag{2.2}$$

  where:

  - m is the total number of faults,

- n is the total number of test cases,
- TF_i is the position of the first test case in the priority order that reveals the fault i.

The other important metrics are:

1. **Effectiveness** or *TPR* or *Recall*: Measures the proportion of actual faults that were correctly identified. A higher *TPR* indicates that the test suite is more effective in finding real bugs.

2. **Precision**: Measures the proportion of test cases predicted as "likely to fail" that actually did fail. High precision minimizes wasted effort in investigating false alarms.

3. **F1-score**: The harmonic mean of precision and recall. It provides a single, balanced metric that is useful when considering both the ability to find real bugs and to avoid false alarms.

4. **Accuracy**: Measures the overall correctness of the test suite's predictions, representing the proportion of all correctly classified cases. Although a general performance measure, it can be misleading with imbalanced datasets, as high accuracy might simply reflect a prediction of the majority class.

5. **Accuracy at** N (or **Top-**N **Accuracy**): This is a crucial business metric, especially in scenarios with high data uncertainty where individual predictions might not be highly confident, or when a single correct label is not enough. Instead of simply looking for the absolute best prediction, the accuracy at N measures how often the true label is present among the top predictions N returned by a model.

   This metric is incredibly valuable in scenarios like recommendation systems, large-scale classification with many possible outcomes, or when dealing with bug fixing and root cause analysis. For example, in bug tracking, if a model suggests the top-N possible teams that might be responsible for a bug, even if the primary suggestion is not correct, having the right team within that top-N can significantly speed up the resolution process by directing attention to other likely candidates.

For a set of predictions:

$$\text{Accuracy at } N = \frac{TP \text{ is among the top-}N \text{ predictions}}{\text{total number of instances}} \tag{2.3}$$

It effectively indicates the system's ability to place the correct answer within a manageable set of top predictions, which can be highly useful in real-world applications where having a few relevant options is often more practical than needing a single perfect one.

Beyond aiming for a definitive class or label in a TSO approach, it is important to measure the change in the shape of the data. Two commonly used metrics to compute geometric relation of datapoints are:

- **Silhouette score** [103] measures how similar a data point (for example, a test case or a software delta) is to its own cluster (group) compared to other clusters. The silhouette score for a single data point i is defined as:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \tag{2.4}$$

  where:

  - $a(i)$ is the average distance between i and all other points in the same cluster,
  - $b(i)$ is the minimum average distance from i to all points in any other cluster (of which i is not a member).

  The silhouette score ranges from -1 to 1, where a higher value indicates better clustering.

- *Davies-Bouldin Index (DBI)* [34] measures the average similarity between each cluster and its most similar cluster:

$$DBI = \frac{1}{N} \sum_{i=1}^{N} \max_{j \neq i} \left(\frac{x_i + x_j}{\text{dist}(i, j)} \right) \tag{2.5}$$

  where:

- N is the number of clusters,
- x_i is the average distance between each point in cluster i and the centroid of cluster i,
- $\text{dist}(i, j)$ is the distance between the centroids of the clusters i and j.

Lower values of DBI indicate better clustering, as it signifies that the clusters are compact and well separated.

2.3 Data landscape of software anomaly in cellular networks

The detection of anomalies in software is crucial and needs a solid grasp of the various data generated during the system's lifecycle. This section explores a variety of usable data types, the key source for detecting anomalies. As shown in Figure 2.7, the SBTS produces and collects a wide range of data, which are then analyzed in the anomaly detection step. Understanding the characteristics of these data types is essential to spot irregularities. Thus, the following subsections will examine binary data, numerical data, categorical data, and textual data, each with specific features and challenges to take into account.

2.3.1 Binary data

In telecommunications, binary data is crucial for capturing and analyzing features of binary nature. These indicators serve as fundamental outcome labels in TSO problems, where each feature essentially represents a success / failure outcome of specific network operations. For example, whether a call was dropped (CD=0/1) or whether a handover was successful (HS=0/1) – all these binary results naturally translate into the overall test result. This binary nature of telecommunications testing creates a hierarchical structure in which individual binary features contribute to the final result. The overall **test result** (TR=0/1) becomes a composite label that aggregates multiple binary results, effectively serving as the primary classification target in the TSO problems. An additional advantage of this method is its ability to easily decompose the problem into a

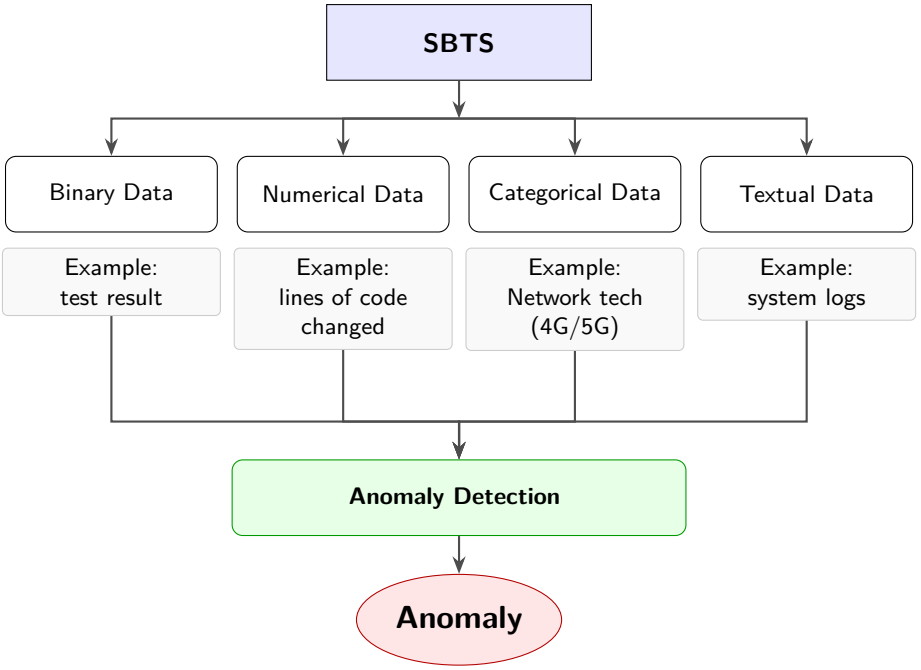


Figure 2.7: The data flow from the variety of produced data, including binary, numerical, categorical and textual types, through its storage, to the anomaly detection step where this data is rigorously analyzed. This crucial analysis, forming the core of this dissertation, culminates in the possible detection of anomalies

multiclass task, such as predicting the outcomes for each individual point. The general binary relationship can be formally expressed as:

$$\mathrm{TR} = function(\mathrm{CD}, \mathrm{HS}, \ldots) = \begin{cases} 1 & \text{if } \bigwedge_{i=1}^{n} (\mathrm{binary\_feature}_i = \mathrm{expected\_value}_i) \\ 0 & \text{otherwise} \end{cases}$$

Now, in telecommunication software testing, several critical network events and conditions are naturally represented as binary outcomes. The binary nature of these features aligns perfectly with the four fundamental classification outcomes (e.g., true positive TP), making them ideal for systematic TSO and performance analysis. For example, a false negative in a handover attempt ($HS =$

0, when it should be 1) could indicate a critical problem in mobility management that requires immediate attention.

- *HO* or **handover success** feature indicates whether a handover between cells or technologies (e.g., 4G to 5G) was successful, as defined in Equation (2.6) below.

$$\mathrm{HO} = \begin{cases} 1 & \text{if } (\mathrm{RSRP}_{\mathrm{target}} > \mathrm{threshold}_{\mathrm{ho}}) \wedge (\tau_{\mathrm{complete}} < \tau_{\mathrm{max}}) \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

  RSRP is short for Received Power of the Reference Signal, used when measuring 4G Long Term Evolution (LTE) networks. A cell phone or other LTE equipped device would display signal strength in RSRP, measured 0 dBm (best signal) to −110 dBm (weakest / no signal).

- *AS* or **authentication success** indicates whether a User Equipment (UE) has successfully completed the security verification process with the network, as presented in Equation (2.7).

$$\mathrm{AS} = \begin{cases} 1 & \text{if } (\mathrm{auth\_response} = \mathrm{valid}) \wedge (\tau < \tau_{\mathrm{timeout}}) \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

- *CD* or **call drop** feature indicates whether a voice call was dropped during the test, as defined in Equation (2.8).

$$\mathrm{CD} = \begin{cases} 1 & \text{if } (\mathrm{call\_state} = \mathrm{terminated}) \wedge (\tau < \tau_{\mathrm{intended}}) \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

The binary nature of features such as handover success, authentication performance, or call dropout status provides straightforward interpretability and easy integration into models. However, the specific nature of this kind of data adds several additional challenges.

The first challenge is a **class imbalance**. Events of interest, such as *CD*, often occur in less than a few percent of cases, leading to a severe skew in the dataset

where the proportion of the minority class is $\ll 0.5$. The imbalance may lead predictive models to favor the dominant class, reducing their accuracy in detecting rare but essential anomalies.

Secondly, the **attribution of causality** is a significant challenge, as single binary outcomes often originate from multiple potential root causes. For example, a *HS* event might arise from poor signal coverage, electromagnetic interference, or underlying hardware problems. This multi-causal nature requires careful and comprehensive feature engineering to capture the contextual information required for accurate diagnosis.

Finally, **verification uncertainty** ($p(\text{true\_label} \neq \text{observed\_label}) > \epsilon$) arises when the ground-truth labels for observed test results become ambiguous or imprecise. In the context of telecommunication software testing, this can manifest itself when a test case indicates a "failure", but the underlying issue is not a software bug, but rather a transient network anomaly, an environmental factor, or a test environment misconfiguration. Such scenarios lead to misclassification, where the observed test outcome does not reflect the true presence or absence of a software defect. This complexity defines a focus of research, particularly for novel classification models, as presented in Section 4.2, that learn the nuanced link between varied data and a binary outcome in heavily imbalanced contexts. Therefore, the objective of the research shifts from the optimization of general precision to emphasize the precision of the minor class, where the detection of true failures in the test result ($TR = 1$) is more critical than the correct identification of passes ($TR = 0$).

2.3.2 Numerical data

Within the scope of this dissertation, several numerical variables are defined to provide a quantitative framework for the analysis of testing processes. To understand the evolution of the code, notable examples of metrics include the change in the lines of code (ΔLoC) and the test execution time:

- **Lines of Code (Δ LoC)**: The number of lines changed in a recent modification, applicable to both tests and deltas. As defined in Equation (2.9), it provides insight into the areas of the code that are actively being devel-

oped or modified. It does not account for the impact or importance of the changes and large changes do not always correlate with higher risk.

$$\Delta\text{LoC} = \lambda(\mathbf{d}) \tag{2.9}$$

where $\lambda : d \to \mathbb{N}_0$, defines the change in Lines of Code (ΔLoC). Here, $\mathbf{d}$ represents a specific code change, such as a commit or a patch, that includes all modifications made. The function λ takes this delta ($\mathbf{d}$) as input and computes ΔLoC as the sum of lines added (A), lines deleted (D), and lines modified (M). This means $\lambda(\mathbf{d}) = A + D + M$.

- **Test Execution Time**: The time required to perform a test, usually measured in seconds or microseconds. This metric indicates whether a test is short or long, but does not provide specific reasons for the execution time. It helps identify tests that may be time-consuming and impact overall testing efficiency. It does not provide context on why the test takes a certain amount of time, and external factors can skew the results.

  For a given test suite $\mathbf{T}$ containing n tests, let $\tau_i \in \mathbb{R}_+$ represent the execution time of an individual test i within that suite. The total execution time, τ_{total}, for the entire test suite $\mathbf{T}$ is defined as following:

$$\tau_{total} : \mathbf{T} \to \mathbb{R}_+, \quad \tau_{total}(\mathbf{T}) = \sum_{i=1}^{n} \tau_i \tag{2.10}$$

These particular metrics are utilized to illustrate key aspects of code evolution and are explained below. Similarly, the outcome of testing activities is often evaluated using measures such as the average execution time of test suites ($\mathbf{T}_{\text{avg}}(\tau)$), which are further correlated with their binary outcomes, serving as a representative example of how the efficiency of testing can be quantified. Then, a feature scaling is applied to each feature individually to ensure that they fall within similar ranges, thus the unit of the feature does not affect learning [113]. From a data point of view, for a matrix with multiple feature-named columns, feature scaling is performed column-wise. The **normalization** scales the numerical features to a specific range, typically [0, 1], which is useful when algorithms are

sensitive to feature scales or when comparing features on a common scale. The **standardization**, also known as z**-score normalization**, transforms the features to have a mean of 0 and a standard deviation of 1, which is helpful for algorithms that assume normally distributed data or when the features have different units.

The scale of features fundamentally influences model training, directly impacting both the learning process and predictive accuracy. Proper feature scaling is therefore essential for algorithms sensitive to the magnitude of features. For example, methods that rely on distance calculations, such as classical machine learning clustering algorithms, as defined in Section 2.4, require standardized feature spaces. Similarly, algorithms employing gradient descent optimizations, demonstrated by neural networks described in Section 2.5, critically require such a standardization to achieve convergence and stable prediction.

Although minimum-maximum normalization is a common approach, it is inadequate for this data because of potential outliers. Instead, a **quantile transformation** is usually used, which transforms the data features to follow a uniform distribution. This method is particularly useful, as it spreads out the most frequent values and reduces the impact of outliers. The transformation is applied independently to each feature (column-wise), mapping the original values to a uniform distribution using the estimated cumulative distribution function. To illustrate the effectiveness of quantile transformation, the conversion of sample data is presented in Table 2.1.

Table 2.1: Data samples before and after quantile transformation

	x_1	x_2	x_3	
Original Data				
Sample 1	1363	28454	38634	...
Sample 2	1394	29055	38956	...
Sample 3	1374	29695	38464	...
Transformed Data				
Sample 1	0.447	0.735	0.614	...
Sample 2	0.455	0.741	0.618	...
Sample 3	0.450	0.747	0.612	...

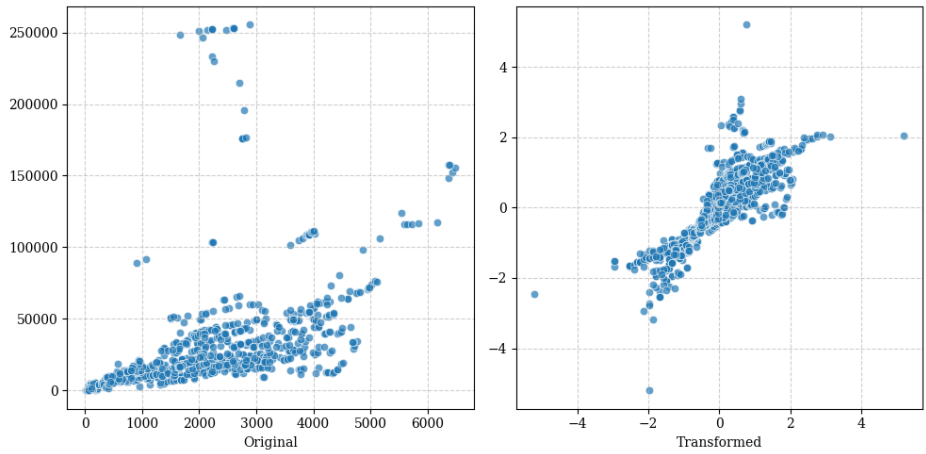


Figure 2.8: Example of quantile transformation effect in relation for two numerical features of the dataset

The quantile transformation effectively normalizes the range of data to follow a specific distribution (normal or uniform), while preserving the relative relationships between the data points and reducing the impact of outliers. This makes it particularly suitable, as it ensures that all features contribute meaningfully to the distance calculations without being dominated by extreme values. Figure 2.8 demonstrates the distribution of data points before and after quantile transformation in the example relation of two features, clearly showing how the transformation achieves a more uniform distribution while maintaining the relative position of the data points.

2.3.3 Categorical data

Categorical data refer to variables whose values are limited to a specific set of defined, non-numeric categories. These categories are *mutually exclusive*, meaning a variable can only belong to one category, *collectively exhaustive*, covering all possibilities, and *qualitative*, indicating types rather than numbers. For example, each cell site in the mobile network data is assigned to a network technology category, ensuring that all sites are included in the classification system.

Below are listed examples of categorical values relevant to both tests and deltas, applicable to the TSO problem:

- **Network Technology** identifies the type of network technology, such as 4G or 5G, which is crucial for the targeted analysis and segmentation of the test results. It may lack granularity for distinctions within 4G or 5G and misses dynamic network performance aspects such as signal strength or user density variations.

  The mathematical representation of this feature, **Network Technology** (NT), shows its possible values as a set of distinct categories:

  $$NT \in \{\text{‘2G’}, \text{‘3G’}, \text{‘4G’}, \text{‘5G’}\}$$

  Each of these categories represents a primary network generation. Within each category, there are various underlying technologies; for example,

  - **2G**: technologies like GSM and EDGE,
  - **3G**: including UMTS and HSPA,
  - **4G**: covering LTE and LTE-A,
  - **5G**: featuring NR SA and NR NSA.

- **Test Scenario** classifies use cases such as voice calls, data transfers, or handovers, providing context for the test results and helping to identify specific scenario issues. It also helps to prioritize crucial scenarios, depending on current needs. It can introduce complexity in data analysis due to the variety of scenarios, and some scenarios may overlap, making it challenging to isolate specific issues.

  The mathematical representation of the feature **Test Scenario** (TS) shows its possible values as a set of distinct scenarios:

  $$TS \in \{\text{‘Voice Call Setup’}, \text{‘Emergency Call (E911)’}, \ldots\}$$

  Each of these values represents a primary test scenario category. These are the specific, high-level test cases that might be executed; for example:

- Voice Call Setup,
- Emergency Call (E911),
- Data Transfer,
- Handover Between Cells,
- Multi-User MIMO.

Each instance of a test would be categorized under one of these predefined scenarios to ensure systematic evaluation of different network functions.

Label Encoding is a basic method that assigns a unique integer to each category. Although memory-efficient because one uses only one column, it can inadvertently introduce misleading ordinal relationships. For instance, assigning '5G' a value of 3 and '2G' a value of 0 might falsely imply that '5G' is numerically "greater" than '2G'. To resolve such ordinal issues, **One-Hot Encoding** creates a distinct binary column for each category. However, this increases data dimensionality; for our four network technologies, it would require four new columns, resulting in a sparse matrix where each row has a single '1' and all other entries are '0's. Striking a balance between label and one-hot encoding, **Binary Encoding** converts categories into binary digits. This approach requires $\log_2(n)$ columns (where n is the number of categories), making it more memory efficient than one-hot encoding, although slightly more complicated to interpret. For four network technologies, one would use just two columns. Finally, **frequency encoding** represents categories by their frequency of occurrence within the data set. This method captures data distribution, which can be useful for certain machine learning tasks but may obscure the original categorical characteristics.

Alternatively, these values can be treated as direct textual data, injected as a part of the sample describing a test or context describing a testing scenario in software anomaly description. This alternative approach is the clue of the research method that focuses on combining multiple textual contexts to perform an accurate prediction, as described later in Section 4.2.

Table 2.2: Anonymized variables reference

Variable	Description
TIMESTAMP	System event timestamp
IDx	Sequential log identifier
PROC-XXX-CEVA	Processor or process identification
UTC_TIME	Time reference in UTC (coordinated universal time)
CORE-x	Processor core number
ADDRx	Memory addresses
LOC_CODE	Error location code
ANONYMIZED_PATH	System file path
SIZE_BYTES	Trace dump size
REG_ADDR	Register address
VALx	Internal state values
HASH_TRACE	Trace verification hash
EO_NAME	Execution object name
EV_GROUP	Event group identifier

2.3.4 Textual data

Unstructured textual data comprises the raw narratives of technical issues encountered daily by testing organizations. Such data, unlike structured formats, resist straightforward categorization and flow continuously within CI processes, representing a blend of technical jargon and casual language, precise error codes, and subjective notes, often describing the same issue in various ways. These narratives, reflecting natural human communication about technical challenges, are rich in context and nuance, offering subtle clues about root causes, indirect links between problems, and key troubleshooting patterns identifiable through contextual analysis. The interpretation of such data is further complicated by the dynamic nature of technology, as exemplified by the introduction of 5G networks, which requires careful consideration of evolving terminologies. For the purpose of this dissertation, the logs and details of the tickets have been anonymized using a systematic obfuscation method, with the nomenclature explained in Table 2.2.

At a fundamental level of complexity, **test and file names** offers the most straightforward method for communicating the essence of their intended purpose. File names, such as `src/system_statistics_em_core_load.c` and `app/dsp/nr/ul/pusch/alg/cfgData/5GMaxRevision.cc`, provide insights into

internal system measurements and low-level uplink functionalities, respectively. Similarly, test names provide insight to specific functionality they relate to, e.g. `General_SiSo_Auto_Real_Time_Energy_Monitoring` and `Interaction between 256QAM and DFT-s-OFDM for TDD PUSCH` relate to performance metrics and specific radio configuration scenarios. As presented, file paths and names often include abbreviations, acronyms, and internal terms. Some abbreviations can be clarified with their extended form, like 3rd Generation Partnership Project (3GPP) documentation, while "alg" and "cfgData" refer to "algorithm" and 'configuration data', respectively. OFDM refers to a specific modulation type, and TDD stands for time division duplex. Each of these abbreviations or acronyms has significant meaning. Easily processed and tokenized, these text pieces have low computational requirements. Although their uniform naming patterns allow for systematic study, they might miss critical language features and require strict compliance with naming conventions.

Then, **system logs** are among the most advanced forms of semi-structured text data and serve as a documentation of complex network systems. Despite initially appearing structured, logs present a web of interlinked information, which complicates standard parsing techniques. These logs reveal narratives about system health, performance issues, and major failures, articulated in a language that merges strict technical protocols with the unpredictable character of real-world system activity. For example, consider a standard LTE network log entry such as the one given below:

```
[TIMESTAMP] [ID3] [PROC-XXX-CEVA] [UTC_TIME] [CORE-2] ERR/FATAL: Fatal
    software exception in LTE DL PHY at [LOC_CODE].
```

This single line, sanitized with sensitive system identifiers and location codes, yet still dense with technical significance, illustrates the complex character of system logs. The difficulty of processing comes from their duality, both structured and unstructured simultaneously. Despite complying with formatting conventions, such as timestamps and standardized identifiers, crucial information is often embedded in free-form messages. Error descriptions, such as *Fatal software exception in LTE DL PHY (...)* combine standardized technical terminology with context-specific details, creating a narrative that requires both domain expertise and advanced parsing techniques to thoroughly process them.

The highest complexity in textual data is the **software anomaly ticket**, which combines various technical narratives, diagnostic information, and context. Unlike the somewhat regular patterns of system logs or test output, software anomaly tickets embody a hybrid documentation style where structured, semi-structured, and unstructured parts co-exist in a complex, interrelated manner. For instance, consider an anonymized example from NOKIA telecommunications infrastructure: a critical anomaly ticket reporting a baseband processor crash during multi-UE traffic testing:

Example of software anomaly description

```
TITLE:
[PRODUCT_VER][PLATFORM][MODE][TECH][HW_TYPE1][HW_TYPE2][HW_TYPE3] crash
    during 4UEs DL/UL traffic testing

TEST STEPS:
Power up base station to on-air state. Connect four test UEs with
    verified correlation parameters between pairs and initiate
    simultaneous UL/DL data traffic streams.

EXPECTED RESULTS:
All four UEs should achieve specified peak throughput values within
    design parameters.

ACTUAL RESULTS:
Baseband processor crash occurred during four-UE concurrent UL/DL traffic
     test execution. Memory pool error detected in processing unit logs.

SYSTEM LOGS:
[TIMESTAMP] [ID1] [PROC-XXX-CEVA] [UTC_TIME] [CORE-2] ERR/PHY: Memory
    Pool Error @ [ADDR1] [ADDR2]
[TIMESTAMP] [ID2] [PROC-XXX-CEVA] [UTC_TIME] [CORE-2] ERR/SYSTEM: Error
    Handler Exception Report
[TIMESTAMP] [ID3] [PROC-XXX-CEVA] [UTC_TIME] [CORE-2] ERR/FATAL: Fatal
    software exception in LTE DL PHY at [LOC_CODE]
[TIMESTAMP] [ID4] [PROC-XXX-CEVA] [UTC_TIME] [CORE-2] ERR/PATH: [
    ANONYMIZED_PATH]/PoolMemUtils.c:155
[TIMESTAMP] [ID5] [PROC-XXX-CEVA] [UTC_TIME] [CORE-2] ERR/TRACE: Dump
    complete, size=[SIZE_BYTES]
[TIMESTAMP] [ID6] [PROC-XXX-CEVA] [UTC_TIME] [CORE-2] ERR/STATE: EO_NAME=
    DlPhyCellEo, REG=[REG_ADDR]
[TIMESTAMP] [ID7] [PROC-XXX-CEVA] [UTC_TIME] [CORE-3] ERR/SYSTEM:
    Concurrent crash detected, ETF stopped
[TIMESTAMP] [ID8] [PROC-XXX-CEVA] [UTC_TIME] [CORE-4] ERR/DRIVER: Current
     state: EO=[VA$L_1$] EV_GROUP=[VAL2]
[TIMESTAMP] [ID9] [PROC-XXX-CEVA] [UTC_TIME] [CORE-4] ERR/TRACE: Trace
    verification [HASH_TRACE]
[TIMESTAMP] [ID10] [PROC-XXX-CEVA] [UTC_TIME] [CORE-5] ERR/SYSTEM:
    Platform fatal error reported by Core 2
```

What makes these tickets particularly challenging for analysis is their inherent narrative complexity. Each section not only carries its own information, but also establishes critical context for interpreting other sections. Thus, pieces of data of this nature are complex due to their layered structure. Tickets start with a precisely formatted title that contains essential classification information, such as hardware types, product version, or the specific platform on which the radio heads were installed. Then a header transition to narrative sections detailing test procedures and expected outcomes, precisely merging plain language with technical specifics. The section `ACTUAL RESULTS` adds complexity, combining factual observations with technical evaluation. The structured system logs document a complex series of cascading events involving multiple components. There is an uncertainty related to such data, where some of the context is hidden behind the front of the data but is known by a human user (for example, specific organization rules or knowledge to which the `EO_NAME=DlPhyCellEo` points to in specific testing scenario). Although the specific structure of this log entry is typical for NOKIA systems, it is important to recognize that other manufacturers and network operators may employ different log formats. Regardless of the particular structure, the fundamental problem of extracting meaning from diverse and complex log data remains a common challenge in the telecommunications industry.

Tokenization is a process of breaking down text pieces (like sentences) into small but meaningful sub-word units (tokens). Tokens serve as basic building blocks for further computational analysis. In the context of software anomaly tickets, effective tokenization is crucial for maintaining semantic meaning while making the text processable by machine learning algorithms. The choice of tokenization strategy impacts how well the system can interpret and analyze different components of these tickets, from standard descriptions of problems written usually in English to specialized technical identifiers such as `[PROC-XXX-CEVA]` or memory addresses. Few commonly recognized techniques are employed across various methods and thus discussed in Chapter 3, with some being the most prominent:

1. **Traditional approaches**

These methods represent fundamental approaches to text segmentation. **Character-level** tokenization divides text into individual characters, offering simplicity and inherent robustness to morphology and spelling variations without requiring pre-training or a vocabulary. However, it generates long sequences and high memory usage, leading to computational inefficiency. Its application dates back to early sequence modeling research, such as in handwriting recognition [51]. In contrast, **Word-level** tokenization treats entire words as tokens, necessitating clear word boundaries and a static vocabulary. Although memory efficient, this traditional Natural Language Processing (NLP) approach struggles significantly with Out Of Vocabulary (OOV) words and lacks adaptability in morphologically complex languages.

2. **Advanced sub-word tokenization models**

   These techniques aim to segment text into sub-word units to mitigate the limitations of word-level approaches, particularly with regard to OOV words and morphological variations. The **Unigram Language Model** is a probability-based technique that optimizes a loss function to maximize the likelihood of data with sub-word units, iteratively removing tokens; it achieves high accuracy but demands considerable computational resources [66]. The **WordPiece** algorithm, popularized by its use in transformer models such as BERT, maximizes the likelihood of vocabulary by constructing sub-words from the bottom up [132]. Complementing these, **SentencePiece** stands out as a language neutral tokenizer that processes raw Unicode sequences directly, eliminating the need for language-specific preprocessing. This flexibility makes it highly suitable for multilingual applications and accommodates various tokenization techniques [67]. Despite the power of their training processes and statistical optimization, these methods can sometimes sacrifice computational efficiency or the clarity of tokenization rules. Consequently, there is often a search for alternatives that strike a better balance between effectiveness and simplicity.

3. **Byte Pair Encoding (BPE)**

   It forms a vocabulary by iteratively merging the most frequent character pairs, presenting a data-driven method that strikes a balance between

character and word level granularity. This approach provides a controllable vocabulary size and efficient data compression. Originally introduced as a data compression algorithm by Gage [42], BPE's application to sub-word tokenization in NLP was popularized by Sennrich, Haddow, and Birch in the context of neural machine translation, where it significantly improved handling of rare words and morphology [108].

Compressing algorithms such as BPE confront the mentioned domain-specific challenges by providing a balanced approach between character- and word-level processing, making it particularly suitable for telecommunications documentation where vocabulary evolution is constant and technical specificity is crucial. The effectiveness of BPE has led to its widespread adoption in state-of-the-art language models, including the Generative Pre-trained Transformer (GPT) family of models [20], and their derivatives (such as Claude [11]), as well as domain-specific models built on the transformer architecture such as CodeBERT [39] and TeleRoBERTa [64]. BPE stands out as an efficient method for handling software anomaly tickets, as it breaks down complex identifiers like "0x8000FF" into parts ("0x", "8000", "FF"). For a better example, let us use the original phrase:

```
Interaction between 256QAM and DFT-s-OFDM for TDD PUSCH
```

The example of a BPE tokenization result could be as follows:

```
Interaction, between, 256, Q, AM, and, D, FT, -s-, OF, DM, for, T, DD, P,
     U, SCH
```

This approach is particularly valuable, as it may preserve the semantic meaning of technical abbreviations (assuming their high frequency in the texts), while reducing the vocabulary size. This is demonstrated by the splitting of compound terms such as DFT-s-OFDM into subcomponents, which are used to decompose other words as well, effectively sharing the same fragments. This capability, when applied to a massive amount of complex, deep, and rich exclusive telecom jargon found in both software anomaly descriptions and system logs, enables an effective form of its representation. Therefore, the BPE algorithm, not only state-of-the-art in LLM, as detailed in Subsection 2.5.1, also serves as a foundation for a new classification model introduced in this PhD thesis and thoroughly discussed in Section 4.2.

2.4 Classical machine learning methodologies

This section examines the core methods of modern machine learning, focusing on how algorithms extract patterns and make predictions from data. Subsection 2.4.1 focuses on the use of unsupervised learning methods to cluster similar data points using different distance metrics. Following this, Subsection 2.4.2 defines several supervised learning techniques to classify data into predefined groups and highlight the essential function of loss functions in model evaluation and optimization. These sections collectively offer an insight into two key machine learning paradigms, providing a benchmarking and baselines method for further research efforts.

2.4.1 Unsupervised models

The practice involves categorizing software tests according to a set of features, such as targeted functionalities, failure patterns, or code coverage. Clustering (grouping) allows for more efficient analysis and management of the test suite by identifying areas that demonstrate similar behaviors or issues. Clustering aims to divide a dataset of $n \in \mathbb{N}$ objects, represented as feature vectors $\mathbf{F} = \{f_1, f_2, \ldots, f_n\}$ where each $f_i \in \mathbb{R}^d$, into k clusters, $K = \{k_1, k_2, \ldots, k_k\}$, in d-dimensional feature space, where $d \in \mathbb{N}$. Ideally, objects in the same cluster should be more similar than objects in different clusters. Similarity (or dissimilarity) is defined through distance metrics, of which the most notable are.

1. **Distance metrics** (L_p norms and others)

   Distance metrics quantify the dissimilarity between data points in a multidimensional space. Among the most commonly used are specific instances of the L_p norm. The **Euclidean distance**, also known as the L_2norm, measures the straight-line distance between two points $\mathbf{x_i}$ and $\mathbf{x_j}$. Mathematically, it is defined below:

$$\text{dist}(\mathbf{x_i}, \mathbf{x_j}) = \sqrt{\sum_l (x_{il} - x_{jl})^2} \tag{2.11}$$

The **Manhattan distance**, or the L_1 norm, calculates the distance as the sum of the absolute differences between coordinates, as provided below.

$$\text{dist}(\mathbf{x_i}, \mathbf{x_j}) = \sum_l |x_{il} - x_{jl}| \tag{2.12}$$

Beyond these common L_p norms, **Hamming distance** quantifies the differing positions between two vectors of identical length, being particularly relevant for binary data. It employs the indicator function $\mathbf{1}(\cdot)$, which returns 1 if its argument is true ($x_{il} \neq x_{jl}$), and 0 otherwise, as follows:

$$\text{dist}(\mathbf{x_i}, \mathbf{x_j}) = \sum_l \mathbf{1}(x_{il} \neq x_{jl}) \tag{2.13}$$

2. **Cosine Similarity and Distance**

The first (**cosine similarity**) measures the cosine of the angle between two non-zero vectors, reflecting their directional alignment rather than their magnitude. This metric is particularly beneficial for tasks like text analysis and in high-dimensional data spaces, where the magnitude of vectors may not be as relevant as their orientation. It ranges from -1 (exactly opposite) to 1 (exactly in the same direction), with 0 indicating orthogonality. The cosine similarity between the vectors $\mathbf{x_i}$ and $\mathbf{x_j}$ is calculated as in Equation (2.14).

$$\cos(\mathbf{x_i}, \mathbf{x_j}) = \frac{\mathbf{x_i} \cdot \mathbf{x_j}}{\|\mathbf{x_i}\|\|\mathbf{x_j}\|} \tag{2.14}$$

where $\mathbf{x_i} \cdot \mathbf{x_j}$ represents the dot product of $\mathbf{x_i}$ and $\mathbf{x_j}$, while $\|\mathbf{x_i}\|$ and $\|\mathbf{x_j}\|$ denote their respective Euclidean norms.

Since clustering algorithms typically require a distance metric (where larger values mean greater dissimilarity), the **cosine distance** is often utilized. This is derived from cosine similarity as its complement, which means that it is 1 minus the cosine similarity, as given in Equation (2.15).

$$\text{dist}(\mathbf{x_i}, \mathbf{x_j}) = 1 - \cos(\mathbf{x_i}, \mathbf{x_j}) \tag{2.15}$$

Hierarchical Agglomerative Clustering (HAC)

It builds a hierarchy of clusters. The agglomerative method, a bottom-up process, begins with individual data points as clusters and merges the nearest clusters using a defined linkage criterion. Popular linkage methods are single (minimum distance), complete (maximum distance), average (mean distance), and Ward's (minimizing within-cluster variance). For example, the single linkage distance between the clusters K_i and K_j is:

$$d(K_i, K_j) = \min_{\mathbf{x_a} \in K_i, \mathbf{x_b} \in K_j} \text{dist}(\mathbf{x_a}, \mathbf{x_b}) \tag{2.16}$$

where $\text{dist}(\mathbf{x_a}, \mathbf{x_b})$ represents a chosen distance metric. Other linkage methods have comparable definitions. The merging continues until all data points consolidate into one cluster, forming a dendrogram representing hierarchical links.

K-means

It seeks to divide the dataset into K predefined clusters, with K being a natural number ($K \in \mathbb{N}$). The K-means algorithm iteratively adjusts the cluster memberships. It starts by randomly setting K cluster centers or centroids, $\mu_1, \mu_2, \ldots, \mu_k$, where each $\mu_i \in \mathbb{R}^d$ lies in the d-dimensional feature space. Each data point $\mathbf{x_i}$ is then linked to its nearest centroid. This linkage is defined as:

$$K_j = \{\mathbf{x_i} \in \mathbf{X} \mid \|\mathbf{x_i} - \boldsymbol{\mu_j}\| \leq \|\mathbf{x_i} - \boldsymbol{\mu_l}\| \text{ for all } l \neq j, 1 \leq l \leq k\} \tag{2.17}$$

Here, K_j represents the j-th cluster, with $\|\cdot\|$ as the chosen distance metric, like the Euclidean distance, which divides the data. The centroids are then updated to the average of points in each cluster:

$$\mu_\mathbf{j} = \frac{1}{|K_j|} \sum_{\mathbf{x_i} \in K_j} \mathbf{x_i} \tag{2.18}$$

Here, $|K_j|$ denotes the count of data points in the cluster K_j. The process of assignment and centroid adjustment continues until convergence, usually marked by minor changes in centroid positions or cluster allocations.

Notably, there exist a K-**medoids** method, which is a sturdier option of K-means and use actual data points, called medoids, as cluster centers, reducing the impact of outliers.

Traditional shallow learning methods function as baseline comparators for the new methodology introduced in this doctoral research. Among the clustering algorithms considered, hierarchical clustering was deemed the most suitable due to its effectiveness and efficiency for this dataset. The initial application aimed to assess the potential of unsupervised grouping to optimize test sets for faster and more cost-effective anomaly detection. This investigation was part of a new methodological approach that integrates two data sources: test results from the telecommunication software and changes to the software itself, as presented in Section 4.1. From there, a hierarchical clustering served as a baseline for the evaluation of supervised learning methods, guiding the design of the custom solution detailed in Section 4.2.

2.4.2 Supervised models

The anomaly detection problem frequently employs supervised learning techniques, often defining the challenge as the classification of test cases based on their relevance to optimization objectives. Although there are various strategies, including ranking test cases by importance, classification methods categorize tests directly into groups relevant for selection or prioritization. The concepts of test prioritization, such as point-wise, pair-wise, and list-wise approaches, can be adapted for classification. The point-wise approach can be viewed as assigning a class label (e.g., “select” or “discard”, “high priority” or “low priority”) to each test case independently. The same supervised models are also widely used in various other classification tasks, for example, the classification of software tickets that describe software anomalies to the proper group of developers for efficient routing and resolution. Detailed in subsequent paragraphs are the specific algorithms often employed, such as logistic regression, Decision Trees, or ensemble methods like XGBoost.

Logistic Regression

Logistic regression models the relationship between the input features and the probability of the outcome using a linear combination followed by a sigmoid function. The features of test cases, such as code coverage and failure history, serve as input for a logistic regression model. This model, indicated by Equation (2.19), describes the relationship between these features and the probability of the given or anticipated label.

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b) \tag{2.19}$$

Here, $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function. It squashes real-valued inputs to a probabilistic range (0–1), allowing for probability interpretation. Then, a softmax function extends the sigmoid function, which is used for binary output, to handle vectors of real values, known as "logits", converting them into probabilities across multiple classes. Each element of the output vector represents the probability of a specific outcome, and all these probabilities sum up to 1. The Softmax function for an input vector of features $\mathbf{F} = [f_1, f_2, \ldots, f_n]$ with possible outcomes $n \in Y$ is defined as:

$$\text{softmax}(f_i) = \frac{e^{f_i}}{\sum_{j=1}^{n} e^{f_j}} \tag{2.20}$$

Once an event's probability is predicted, the model must assess how accurate these predictions are. This assessment uses a loss function that measures the difference between output probabilities and true labels. In binary classification tasks, distinguishing between two outcomes, Binary Cross-Entropy (BCE) loss, as defined in Equation (2.21), is widely used.

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{2.21}$$

where:

- J is the calculated **loss (or cost)**, reflecting the model's performance given its current parameters.
- $\boldsymbol{\theta}$ represents the **model's parameters** (e.g., weights and biases). The loss J

directly depends on the model predictions $\hat{y}_i$, which are, in turn, parameterized by θ. Optimization aims to find θ that minimizes J.

- n is the total **number of training examples**.
- $y_i \in \{0, 1\}$ is the **true binary label** for the i-th example (e.g., 0 for "normal", 1 for "anomaly").
- $\hat{y}_i \in [0, 1]$ is the **predicted probability** for the i-th example, typically derived using a sigmoid activation function: $\hat{y}_i = \sigma(z_i)$.

Decision Trees

The aim of a Decision Tree is to create a tree-like structure to classify the data based on a series of decisions made about the features. The tree is built by recursively splitting the data based on the feature that provides the most information gain or reduces impurity the most. The main parameter is the maximum depth, which limits the depth of the tree to prevent overfitting. Instead of a loss function, Decision Trees use impurity measures. Then, **Random Forests** create an ensemble of Decision Trees, each trained on a random subset of data and a random subset of features. The final prediction is made by aggregating the predictions of all trees (e.g., using majority voting). The user defines the number of trees to be generated and their maximum depth.

In the construction of the Decision Tree, the entropy and Gini impurity metrics guide the data splits at each node, aiming to measure sample "impurity" or "disorder". They seek to identify the feature that produces the most homogeneous child nodes, thus maximizing information gain. If c is the number of classes and p_c is the proportion of samples belonging to the class , then we can define both ideas in the following paragraphs.

Entropy measures the average information needed to identify the label of the class of a sample, as defined in Equation (2.22). In particular, entropy might be a little slower to compute than Gini [100].

$$\text{entropy}(p) = -\sum_{k=1}^{K} p_k \log_2(p_k) \tag{2.22}$$

Gini impurity or gini(p), represents the probability of misclassifying a randomly chosen element if it were randomly labeled according to the class distribution.

$$\text{gini}(p) = 1 - \sum_{k=1}^{K} p_k^2 \tag{2.23}$$

XGBoost

This model builds trees iteratively, each tree correcting the errors of the previous trees. It uses gradient descent to minimize a regularized objective function. Parameters are similar to Decision Trees, controlling the complexity of individual trees. The main parameters are the number of estimators (trees), max depth as well and learning rate, which controls the contribution of each tree to the overall prediction. XGBoost uses a regularized objective function that combines a loss function (e.g., BCE) with a regularization term:

$$\text{Obj} = \sum_{i=1}^{n} J(y_i, \hat{y}_i) + \sum_{m=1}^{M} \Omega(e_m) \tag{2.24}$$

Here, J is the loss function, $\hat{y}_i$ is the predicted value, e_m is the m-th tree, and Ω is the regularization term.

Following mentioned methods, Section 2.5 defines several more sophisticated learning techniques to classify data into predefined groups and highlight the essential function of loss functions in model evaluation and optimization. However, these classical methods, critical to serving as benchmarking and baseline comparators, were instrumental in evaluating subsequent innovations and established the fundamental insights critical to designing and developing the novel and efficient classification approach detailed in Section 4.2.

2.5 Neural networks

The prior section examines classical and shallow machine learning models. While models such as logistic regression and Decision Trees are well established, their

performance may decline with complex datasets due to non-linear relationships and hidden structures. Neural Networks (NN) address these gaps by dynamically learning complex patterns, adjusting connection of weights and biases among artificial neurons. This fundamental capability to model non-linear relationships makes NNs particularly well-suited for analyzing complex scenarios. For example, in telecommunication systems, they can effectively identify subtle indicators of potential failures embedded within extensive but semantically rich system logs or identify deviations within time-series performance data indicated by different numerical metrics over time.

Before applying the loss functions, the raw outputs of the network (logits) are transformed into appropriate probability distributions using the **softmax activation function**. The softmax function, previously defined in Equation (2.20) takes a vector of arbitrary real-valued scores and transforms them into a probability distribution over the predicted output classes. This transformation ensures that all probabilities are positive and sum to 1 in all classes, making them valid probability estimates for classification tasks.

Then, the intuition of BCE is that if $y_i = 1$ (true label is positive), the term $(1-y_i)\log(1-\hat{y}_i)$ becomes zero, and the loss is dominated by $-\log(\hat{y}_i)$. We want $\hat{y}_i$ to be close to 1 to minimize the loss. But if $y_i = 0$ (true label is negative), the term $y_i \log(\hat{y}_i)$ becomes zero, and the loss is dominated by $-\log(1-\hat{y}_i)$. We want $\hat{y}_i$ to be close to 0 to minimize the loss. In particular, a BCE loss is a special case of Categorical Cross-Entropy (CCE) when there are only two classes ($K = 2$).

The **CCE** loss, as defined in Equation (2.25), is used for multi-class classification problems, where the goal is to predict one of several possible outcomes (e.g., classifying images into different categories).

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_{ik}\log(\hat{y}_{ik}) \tag{2.25}$$

where:

- J is the calculated **loss (or cost)**, reflecting the model's performance given its current parameters.

- θ represents the **model's parameters** (e.g., weights and biases). The loss J

directly depends on the model predictions $\hat{y}_{ik}$, which are, in turn, parameterized by θ. Optimization aims to find the θ that minimizes J.

- m is the total **number of training examples**.
- K is the total **number of distinct classes**.
- $y_{ik} \in \{0, 1\}$ is the **true binary label** for the i-th example and the k-th class. This is typically a one-hot encoding, meaning $y_{ik} = 1$ if the i-th example belongs to class k, and $y_{ik} = 0$ otherwise.
- $\hat{y}_{ik} \in [0, 1]$ is the **predicted probability** for the i-th example belonging to class k, calculated using the softmax function defined in Equation (2.20).

If we work with hard coded probabilities, it means that only one y_{ik} will be 1 (the true class) and probability is not spread across vector, and all others will be 0. The loss function then simplifies to:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \log(\hat{y}_{i,\text{true\_class}}) \tag{2.26}$$

Therefore, we want the predicted probability for the true class $\hat{y}_{i,\text{true\_class}}$ to be as close to 1 as possible to minimize loss.

Eventually, the CCE has become a cornerstone in the training of NNs, particularly in LLMs, described in Section 2.5.1, where it effectively measures the difference between predicted probability distributions in multiple classes of tokens. In these models, CCE helps optimize the network's ability to predict the next token in a sequence by minimizing the difference between the model's predicted probability distribution and the actual distribution of the target tokens. Then, BCE, as its binary counterpart, finds its crucial application in the proposed LMLDA model presented in Chapter 4.2. The model uses BCE as its primary loss function to learn the binary classification task of predicting whether a given test and the combination of software deltas will result in a bug discovery. This binary formulation allows the model to effectively learn and encode complex patterns from extensive telecommunication testing data, where the outcome is inherently binary, that is, a bug must be discovered or not. This application

demonstrates how fundamental loss functions can be adapted from general machine learning principles to solve specific industry-relevant problems in software testing and quality assurance.

2.5.1 Learning-based text representations

Traditional methods, often characterized as **shallow learning** techniques, function as baseline comparators for the new methodology introduced in this doctoral research, particularly in contrast to modern, deep models such as LLM.

The evolution of advanced text analysis in telecommunications originates from initial neural network models aimed at interpreting local patterns. A foundational example is **FastText** [18], which provides a strong basis for text representation. Effective **anomaly detection** in 4G and 5G telecom systems depends on correctly interpreting large volumes of **textual data**, which is a hybrid mix of natural language (filled with industry-specific abbreviations and jargon) and structured technical strings like error codes, configuration parameters, and alphanumeric identifiers. In telecommunications, where reliability and trust are critical, sophisticated NLP techniques are vital for analyzing such a complex data. This section examines the transition from these initial local context models to advanced models LLM, forming its theoretical foundation.

The evolution of advanced text analysis in telecommunications originates from initial neural network models aimed at interpreting local patterns. A foundational example is **FastText** [18], which efficiently generates vector representations of words ($\mathbf{v}_w$) as a sum of its constituent character vectors n-gram (g_i). Formally, for a word w composed of n-grams $g_1, g_2, \ldots, g_n$, its representation is given by the following equation:

$$\mathbf{v}_w = \sum_{i=1}^{k} \mathbf{v}_{g_i} \tag{2.27}$$

where $\mathbf{v}_{g_i}$ is the vector for the i-th n-gram.

To describe the advantage of such a method, let us consider a theoretical scenario in which during a testing, the anomaly was identified as `PHY_SYNC_FAIL` on slot `ID_5G_FR1_TX_001`. For a technical string such as `PHY_SYNC_FAIL`, FastText

can generate a meaningful embedding by combining n-grams such as `PHY`, `HY_`, `Y_S`, allowing a potentially unknown words to be represented by smaller parts, which can be constructed into the normally OOV word. This capability is vital for handling such terms, where full-word embeddings often struggle due to data sparsity or their OOV nature. FastText's ability to model the morphological aspects of language and address OOV challenges represents a significant, although limited, step in building robust linguistic representations for specialized domains.

Despite these advancements, models like FastText, mainly relying on fixed-size local contexts, faced fundamental limitations when processing long and complex sequences of telecommunications logs and event streams. Earlier sequential models, particularly standard Recurrent Neural Network (RNN), which unlike traditional feedforward NN, are designed to process sequences, faced a significant challenge known as the problem of vanishing gradients [53]. This problem significantly impeded their ability to effectively model long-term dependencies. Gradients propagated over multiple time steps would diminish exponentially. Consequently, it caused critical information from early in a sequence to degrade or disappear by later processing stages. To address this fundamental limitation, the same author proposed the Long-Shor Term Memory (LSTM) network [54]. LSTMs employ three types of multiplicative gates –- forget, input, and output gates – to regulate the information flow. The forget gate determines what information to discard from the memory cell, the input gate controls what new information is stored, and the output gate decides which information from the cell state is exposed as the hidden state for subsequent steps. However, the vanishing gradient problem becomes observed once more, with the eventual need to process much longer passages of textual data.

To address these constraints, the **attention mechanism** was introduced [10, 80]. In contrast to n-grams that rely on fixed local contexts, attention lets the model dynamically weigh the importance of tokens throughout a sequence. Formally, as defined in Equation (2.28), the attention function calculates a weighted sum of the value vectors ($\mathbf{V}$), where the weights are determined by the similarity between a query ($\mathbf{Q}$) and the key vectors ($\mathbf{K}$) from the input sequence.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{2.28}$$

Here, $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$ represents the **Q**uery matrix, $\mathbf{K} \in \mathbb{R}^{m \times d_k}$ is the **K**ey matrix, and $\mathbf{V} \in \mathbb{R}^{m \times d_v}$ is the **V**alue matrix. The d_k is a scaling factor crucial for preventing vanishing gradients.

This approach lets the model focus on the entire input sequence, effectively tackling the vanishing issue seen in earlier architectures. To illustrate this, consider a more comprehensive telecommunications log sequence:

```
[2024-06-07 10:00:01] [CRITICAL] [PHY_LAYER] ERROR: PHY_SYNC_FAIL on
    ID_5G_FR1_TX_001.

Reason: DL_SIGNAL_LOSS.
Counters: N_SYNC_ATT=100.
Previous event: [2024-06-07 09:59:45] [WARNING] [RF_MODULE]
    RF_PWR_DEVIATION_HIGH.

Operator note: Repeated issue after yesterday's config change
```

An attention-equipped model can now effectively link `PHY_SYNC_FAIL` not only with the immediately preceding `DL_SIGNAL_LOSS` but also with the distant `RF_PWR_DEVIATION_HIGH` (a potential precursor) and the even more remote `yesterday's config change`. This capability to identify non-local, yet semantically critical, dependencies is absolutely fundamental for precise anomaly detection, as complex network issues often manifest as subtle combinations of events scattered throughout vast, heterogeneous data streams.

The pivotal innovation consisting of introduction of attention led directly to the development of the transformer architecture [124], which abandoned recurrence in favor of stacking of attention layers. Modern LLM builds directly on the transformer and its **Multi-Head Attention**. The mechanism is defined in Equation (2.30). For given queries Q, keys K, and values V, and with h attention heads, Multi-Head Attention is computed as:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, ..., \text{head}_h)\mathbf{W}^O \tag{2.29}$$

Each head_i independently processes the relationships between tokens by applying the learned linear transformations $\mathbf{W}_i^Q, \mathbf{W}_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ and $\mathbf{W}_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ to queries, keys, and values before computing attention, as defined in the following:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{2.30}$$

Each head_i independently applies linear transformations to the input matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ using their respective learned weight matrices $\mathbf{W}_i^Q$, $\mathbf{W}_i^K$, and $\mathbf{W}_i^V$. The resulting products, such as $\mathbf{Q}\mathbf{W}_i^Q$, serve as inputs to the scaled dot-product attention function. The concatenated outputs are then further linearly transformed by $W^O \in \mathbb{R}^{hd_v \times d_{model}}$. This multi-headed approach allows the model to respond to information from different representation subspaces and capture diverse relational patterns across various positions within the input sequence.

Therefore, LLMs are deep neural networks and probabilistic systems that learn to predict the likelihood of a subsequent token given its preceding context. This autoregressive nature aligns with the fundamental **chain rule of probability**, defined in Equation (2.31), where the joint probability of a sequence of events $s_1, s_2, \ldots, s_n$ is decomposed into a series of conditional probabilities.

$$p(s_1, s_2, \ldots, s_n) = \prod_{i=1}^{n} p(s_i | s_1^{i-1}) \tag{2.31}$$

LLMs are trained to estimate these conditional probabilities. The primary training objective typically involves minimizing the CCE loss, which is equivalent to maximizing the log-likelihood of the training data. For a given sequence of tokens $\mathbf{s} = (s_1, s_2, \ldots, s_n)$, the training loss function is formalized as in Equation (2.32)

$$\ell_{\text{CE}} = -\frac{1}{N} \sum_{n=1}^{N} \log p_\theta(s_n | s_{<n}) \tag{2.32}$$

Here, $p_\theta(s_n | s_{<n})$ denotes the predicted probability of the model for the n-th token s_n, conditioned on the preceding tokens $\mathbf{s}_{<n} = (s_1, \ldots, s_{n-1})$, and $\boldsymbol{\theta}$ encompasses all the learnable parameters. Minimizing this loss forces the model to assign high

probabilities to the actual next tokens observed in the training data, thereby optimizing its predictive accuracy.

The performance of an LLM is frequently assessed using **perplexity** (PPL) [58]. As defined in Equation (2.33), for a given tokenized sequence $X = (s_1, s_2, \ldots, s_n)$, perplexity is mathematically defined as the exponential of the average negative log-likelihood of the sequence.

$$PPL(X) = \exp\{\ell_{\text{CE}}\} = \exp\left\{-\frac{1}{N}\sum_{n=1}^{N}\log p_\theta(s_n|s_{<n})\right\} \tag{2.33}$$

A lower perplexity score indicates greater confidence and accuracy in the model's predictions, suggesting a better understanding of the underlying data distribution. While perplexity is highly dependent on training data and tokenization scheme, making direct comparisons between different models or datasets difficult without careful consideration, this metric remains a valuable tool for evaluating a model's performance on a single, well-defined task. As noted by Meister et al. [87], this dependency requires careful attention to the experimental setup to ensure meaningful results.

The unique aspect of LLMs is their immense scale, with billions of parameters and training datasets spanning petabytes of varied text and code. This scale, coupled with the efficient self-attention mechanism, results in the increase in emerging abilities such as advanced reasoning, deep contextual understanding, and few-shot learning [20] – the ability to learn a new task from very few examples without explicit fine-tuning. Furthermore, these contextual embeddings enable complex reasoning patterns, exemplified by chain-of-thought prompting [130], where LLMs can articulate intermediate reasoning steps for problem-solving. In the context of this doctoral thesis, focused on anomaly detection in 4G and 5G base station programming, these advanced capabilities of LLMs are invaluable. They allow a thorough analysis of system logs, spot unusual patterns in complex software setups, and aid in routing software tickets to the proper developer group, as later presented in Section 5.3. However, directly applying these broadly pre-trained models to highly specialized and often proprietary domains, such as telecommunications operations, presents unique challenges related to data specificity, accuracy, and currency. To effectively deploy LLMs in such niche

contexts, two primary strategies are employed: Retrieval-Augmented Generation (RAG) [72] and fine-tuning, often utilizing parameter-efficient methods like Parameter-Efficient Fine-Tuning (PEFT) [37] and its prominent variant, Low-Rank Adaptation (LoRA) [55].

RAG is a powerful approach, which enhances the generative capabilities of the LLM by providing it with domain-specific external information at inference time [73]. The process typically involves an initial retrieval step, where a user's query or a specific context from a telecommunications system (e.g., a critical error code, a description of network traffic anomaly) is used to search a specialized knowledge base. This knowledge base might include internal documentation, past incident reports, detailed protocol specifications, or proprietary log schemas. The most relevant documents or text chunks ($\mathcal{D}$) identified by the retrieval system are then concatenated with the original query, forming an augmented input for the LLM. The LLM then generates its response conditioned on this combined, grounded information. Mathematically, this shifts the conditional probability from solely $p(s_n|s_{<n})$ to $p(s_n|s_{<n}, \mathcal{D})$, where $\mathcal{D}$ represents the retrieved context, effectively allowing the LLM to base its predictions not just on its pre-trained general knowledge, but also on real-time, accurate, and domain-specific facts. In the telecommunications sector, specific frameworks such as Telco-RAG are being developed to demonstrate the utility of RAG in telecommunications by leveraging domain-specific knowledge bases for accurate and relevant responses [19].

Alternatively, **fine-tuning** involves adapting a pre-trained LLM to a specific downstream task or domain by updating a small subset of its original parameters. Although the complete fine-tuning of multi-billion-parameter LLMs is computationally intensive and requires large, labeled domain-specific datasets, PEFT methods have emerged to mitigate these challenges. For example, LoRA operates by freezing the vast majority of the original pre-trained weight matrices (W_0) within the transformer layers and instead injects small, trainable rank-decomposition matrices. For a given weight matrix $W_0 \in \mathbb{R}^{d\times k}$ in a layer, LoRA adds a low-rank update $\Delta W = BA$ such that the new weight matrix becomes $W_0 + BA$. Here, $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, with $r \ll \min(d, k)$. This dramatically reduces the number of trainable parameters from $d \times k$ to $d \times r + r \times k$, making fine-tuning feasible on modest hardware and with smaller domain-specific

datasets. In the context of telecommunications, fine-tuning an LLM allows it to deeply internalize domain-specific terminology, recognize intricate patterns unique to telecom logs, or generate highly specialized prose for anomaly reports [56]. For example, fine-tuned LLMs can be used to automatically summarize network events, classify nuanced error types, or even generate configuration scripts that adhere to specific operational guidelines.

This type of modeling motivated the research efforts thoroughly elaborated in Chapter 5. This dissertation introduces new research targeting two main goals: first, to assess the correlation between improvements in the perplexity metric, reflecting the learning of new token distributions, and subsequent changes in the model's classification performance. Second, it aimed to determine the behavior of LLM in situations with normally unknown confidence of the prediction. This is especially relevant given the challenging environment of telecommunication data, which is filled with exclusive vocabulary and a token distribution that diverges significantly from the original corpora on which LLM are typically trained. The unique jargon, acronyms, and technical specifications characteristic to telecommunications datasets present a substantial challenge for pre-trained models, as their internal representations of language may not adequately capture the nuances and specific relationships within this specialized domain. Eventually, the study examines how LLMs adapt to such novel token distributions and whether a reduction in perplexity within this specialized context directly translates into improved classification accuracy.

2.6 Probabilistic approach

The chapter begins with a brief overview of Bayes' theorem and its role in the naive Bayes classifier, establishing a basic probabilistic framework. Following this, the concept of BNN is examined, which uses probabilistic priors on weights to quantify the uncertainty of the model. To address computational difficulties in complex probabilistic models, variational inference VI is introduced as a method to approximate challenging posterior distributions, making these models feasible. The discussion then progresses to topic modeling, focusing mainly on LDA as a central method to identify hidden patterns in text data. Building on previous

content (see Section 2.2.1), the chapter further explores uncertainty quantification, distinguishing between aleatoric uncertainty from inherit model and data randomness, and epistemic uncertainty from model knowledge gaps due to limited training data. This detailed analysis is crucial for measuring and improving the reliability of predictive models, which is essential for reliable decision-making in complex systems.

2.6.1 Bayes theorem and its application

A Bayes theorem is a fundamental principle in probability theory that provides a framework for updating the probability of a hypothesis in light of new evidence. It precisely defines the conditional probability of an event, given that another event has occurred. Mathematically, for any two events e_1 and e_2, Bayes theorem states:

$$p(e_1|e_2) = \frac{p(e_2|e_1)p(e_1)}{p(e_2)} \tag{2.34}$$

Here, $p(e_1|e_2)$ is the *posterior probability* (the probability of hypothesis e_1 given evidence e_2), $p(e_2|e_1)$ is the *likelihood* (the probability of evidence e_2 given hypothesis e_1), $p(e_1)$ is the *prior probability* of hypothesis e_1 (the initial probability before considering evidence), and $p(e_2)$ is the is the *marginal probability* or *probability of the evidence*.

The naive Bayes classifier provides a practical application of the theorem. Its objective is to find the class C_n with the highest posterior probability $p(C_n|\mathbf{F})$ given the observed features $\mathbf{F}$. The core of a Bayesian classifier is the maximum posterior estimation, as defined in the following equation:

$$\hat{y} = \arg\max_{C_n} P(C_n|F) \tag{2.35}$$

This simplifies to $\hat{y} = \arg\max_{C_n} p(\mathbf{F}|C_n)p(C_n)$ because $p(\mathbf{F})$ is constant for all classes. The "naive" assumption of conditional independence between features f_i given the class C_n is then introduced. This crucial simplification allows the likelihood $p(\mathbf{F}|C_n)$ to be expressed as a product of individual conditional proba-

bilities, making the model computationally efficient. The classifier is then defined as follows:

$$\hat{y} = \arg\max_{C_n} p(C_n) \prod_{i=1}^{n} p(f_i|C_n) \tag{2.36}$$

Prior probabilities $p(C_n)$ and individual likelihoods $p(f_i|C_n)$ are estimated from training data, typically as frequencies for categorical features such as developer experience (e.g., counting anomalous changes by developers) or from assumed distributions for continuous features such as changed lines of code (e.g., applying a Gaussian distribution to assess line of code in delta between anomalous and non-anomalous changes in software delta).

2.6.2 Bayesian Neural Network (BNN)

BNN represent a fundamental extension of traditional NN by treating network weights (w, while $w \in \theta$) as probability distributions rather than fixed-point estimates [17]. This Bayesian approach provides a principled framework for performing inference over model parameters, which is crucial for capturing and quantifying the uncertainty inherent in the model predictions. In safety-critical applications such as telecommunications, where overconfident predictions can lead to significant service disruptions, this ability to assess prediction reliability is indispensable.

In a BNN, the goal is to infer the posterior distribution over the weights of the network (w) given the training data, for example a set of textual documents, observed and defined as $\mathbf{D} = \{\mathbf{X}, \mathbf{Y}\}$. According to Bayes' rule, this posterior ($p(w|\mathbf{D})$) is expressed as in the following equation.

$$p(w|\mathbf{D}) = \frac{p(\mathbf{D}|w)p(w)}{p(\mathbf{D})} = \frac{p(\mathbf{Y}|\mathbf{X}, w)p(w)}{\int p(\mathbf{Y}|\mathbf{X}, w')p(w')dw'} \tag{2.37}$$

where:

- $p(w|\mathbf{D})$ is the posterior distribution on the weights, reflecting our updated belief after seeing the data D.

- $p(\mathbf{D}|w) = p(\mathbf{Y}|\mathbf{X}, w)$ is the likelihood function, representing the probability of observing the data $\mathbf{D}$ given a specific set of weights w.
- $p(w)$ is the prior distribution on the weights, encoding any initial beliefs or regularizations (e.g. to prevent overfitting). A common choice for $p(w)$ is a Gaussian distribution of zero mean, favoring smaller weights and encouraging smoother functions.
- $p(\mathbf{D})$ is the evidence of the model (or marginal likelihood), a normalization constant that is often intractable to compute directly, as it involves integration over all possible weights.

It is necessary to consider how the posterior distribution of a single hypothetical weight w evolves during training. Initially, the belief is broad, represented by the prior distribution. As the model processes more data (or progresses through more training epochs), the posterior distribution becomes increasingly concentrated around values of w that are highly consistent with the observed data. This process of updating and narrowing the belief, reflecting increasing certainty, is conceptually illustrated in Figure 2.9.

Variational Inference (VI)

Due to the intractable nature of the true posterior distribution $p(w|\mathbf{D})$ for most complex BNN architectures, exact Bayesian inference is computationally demanding or even impractical. Therefore, approximate inference techniques are employed, with VI being a widely adopted method [17, 44]. The VI aims to approximate the true posterior $p(w|D)$ with a simpler and more tractable distribution, denoted as $q_\theta(w)$, parameterized by $\boldsymbol{\theta}$. Here, θ represents the trainable parameters (e.g. means and variances) of the approximate distribution over the weights of a Bayesian layer, such as the variational dense layer, as defined in the following equation:

$$q_\theta(w) \approx p(w|\mathbf{D}) \tag{2.38}$$

This approximation is achieved by minimizing the Kullback-Leibler Divergence (KLD) between the approximate posterior $q_\theta(w)$ and the true posterior

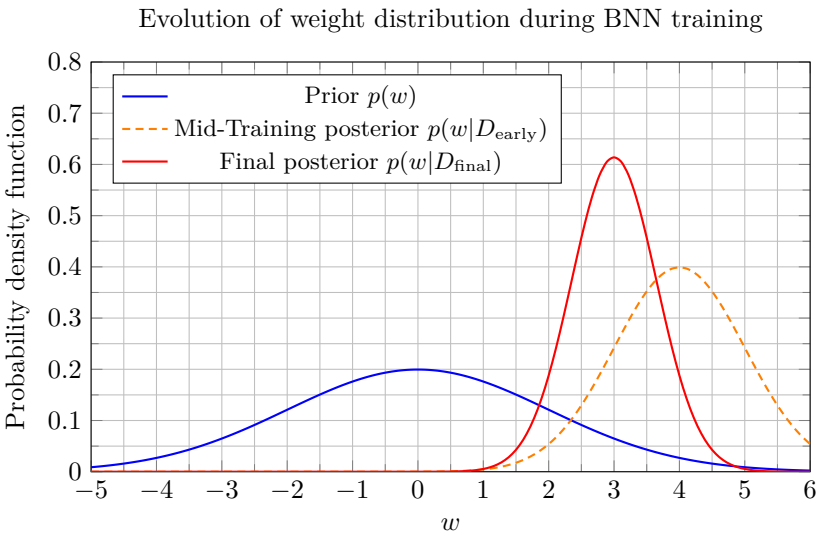


Figure 2.9: Evolution of the belief about a single neural network weight (represented by its probability distribution) through different stages of BNN training. The distribution becomes progressively narrower and more precise as more data is observed

$p(w|\mathbf{D})$. Minimizing the KLD is equivalent to maximizing a lower bound on the model evidence, presented in Equation (2.39) and often referred to as the Evidence Lower Bound (ELBO).

$$\mathcal{L}(\theta) = \mathbb{E}_{q_\theta(w)}[\log p(\mathbf{D}|w)] - \mathrm{KLD}(q_\theta(w)||p(w)) \tag{2.39}$$

The first term, the expected log-likelihood, quantifies how well the approximate posterior's sampled weights fit the observed data $\mathbf{D}$. The second term, the KLD divergence between the approximate posterior and the prior, acts as a regularizer, preventing $q_\theta(w)$ from deviating too far from our initial beliefs about weights. This regularization is crucial for avoiding overfitting, as discussed previously. This allows network operators to differentiate between a high-confidence prediction of severe degradation versus a low-confidence prediction that might warrant further investigation before intervention. In alignment with the research presented in Section 5.4 on software anomaly classification, BNNs can provide de-

velopment teams with not just a predicted anomaly type, but also the confidence associated with that classification.

Applying BNN principles to LLM fine-tuning involves replacing the deterministic classification head with a variational layer, trainable while the LLM backbone is often frozen (treating LLM as encoder). This approach naturally mitigates overfitting because the prior distribution over weights (e.g. a zero-mean Gaussian) acts as a regularizer, favoring simpler solutions and balancing data fit with prior beliefs. Critically, by sampling multiple weight configurations from the learned posterior of this variational layer and generating the corresponding predictions, one can effectively quantify epistemic uncertainty, revealing the confidence of the model in its output. This forms the basis of the novel research described in Chapter 5.

Uncertainty quantification

Quantifying uncertainty in predictions is critical to building reliable and trustworthy systems, especially in high-stakes domains such as telecommunications [65]. It allows for informed decision making, differentiating between model confidence and inherent data variability. Bayesian methods naturally decompose total predictive uncertainty $\omega^2_{\text{total}}(x)$ into two distinct components: aleatoric uncertainty and epistemic uncertainty. These were briefly introduced in Section 2.2.1, presenting sources for both types, thus the following paragraphs will concentrate on modeling characteristics of these two variability types.

Aleatoric uncertainty originates from the inherent randomness in data or phenomena. This uncertainty remains unaffected by additional data or model enhancements because it involves irreducible randomness. For a model's output y with input x, **aleatoric uncertainty**, as shown in Equation (2.40), is generally represented by the expected prediction variance averaged over viable model weight configurations.

$$\omega_a^2(x) = \mathbb{E}_{q_\theta(w)}[\omega^2(x, w)] \tag{2.40}$$

The $\omega^2(x, w)$ is the noise variance modeled by the network for input x and weights w. In telecommunications, examples include sensor noise from base sta-

tions, random fluctuations in radio propagation channels leading to variable signal quality measurements. In ML models, particularly neural networks, this often concerns the pseudo-random initial weights used to start the training. Even a carefully designed and initialized model would still struggle with this level of uncertainty.

Epistemic uncertainty (also known as model uncertainty or reducible uncertainty) originates from the model's lack of knowledge about the true underlying function. This uncertainty is primarily due to limited training data $\mathbf{D}$ or limitations in the architecture of the model. Unlike aleatoric uncertainty, epistemic uncertainty can theoretically be reduced by collecting more data, selecting a more appropriate model (or parameters), or employing more effective inference techniques. The $\omega_e^2(x)$, as defined in Equation (2.41) is estimated via the variance of predictions across different plausible models (or weight configurations) learned by the Bayesian network.

$$\omega_e^2(x) = \mathrm{Var}_{q(w)}[\mu(x, w)] \tag{2.41}$$

where $\mu(x, w)$ is the mean prediction of the network for the input x with weights w.

In the context of the variational dense layer used for the LLM fine-tuning, this is practically estimated by performing Monte Carlo sampling of the layer's weights from its learned posterior distribution and computing the variance across the resulting multiple predictions for the same input. In a telecom context, high **epistemic uncertainty** might arise when predicting network performance in novel or rarely observed operating conditions, detecting new types of anomalous behavior in system logs for which limited training data exist, or evaluating system behavior in configurations that were under-represented in the training set.

Then, the **total predictive uncertainty**, as defined in Equation (2.42), for a given input x combines both sources of uncertainty, providing a comprehensive view of the model's confidence in its output.

$$\omega_{\text{total}}^2(x) = \omega_a^2(x) + \omega_e^2(x) \tag{2.42}$$

Understanding both components of uncertainty is crucial for operational

decision-making in telecommunications, especially for sophisticated models like LLM. For example, if the measured value is high and predominantly epistemic, it signals that the model is "uncertain" due to lack of knowledge. In the context of LLMs, this means that certain data types or features are missing or underrepresented. This insight can lead to collecting more relevant data, resampling existing data, or expert intervention to improve the model's performance. If the uncertainty is predominantly aleatoric, it indicates inherent noise in the system that must be managed, perhaps through an improved model architecture or a different set of parameters rather than simply more data. When an LLM is used for predictions, high uncertainty acts as a warning, allowing users to know when a prediction might not be reliable and may require a human to double check it. This ability to measure confidence also benefits basic research by allowing us to study how an LLM's design affects its ability to express certainty, which can point out areas where more knowledge is needed. These findings are directly relevant to the core research presented in Section 5.4.

2.6.3 Topic Modeling

The Naive Bayes classifier assumes strict conditional independence of features given the class, which simplifies the computation of posterior probabilities for classification. Although useful, this assumption often misses the inherent correlations and thematic patterns in complex datasets, especially text, a shortcoming that more sophisticated Bayesian methods like BNN and VI aim to address by modeling richer dependencies and estimating complex posteriors. While BNN and VI apply Bayesian concepts to complex model structures for tasks such as discriminative prediction and uncertainty quantification, latent Dirichlet allocation LDA provides a probabilistic framework to identify hidden thematic structures (or topics) within a text corpus [16]. It models documents as mixtures of these topics, each topic defined as a probability distribution over words. Unlike hard clustering methods, presented in Section 2.4.1, which assign documents to single discrete group, LDA's topics function as soft clusters. A document can belong to multiple topics simultaneously, each with a specific probability. Similarly, a topic is not just a collection of words, but a probability distribution over the entire vocabulary, reflecting how relevant each word is to that topic. This inher-

ent probabilistic nature, which extends to both document-topic and topic-word relationships, distinguishes LDA from traditional clustering by offering a more nuanced and transparent understanding of thematic organization of a corpus.

Mathematical formulation of LDA

The model assumes a generative process in which documents are created by sampling topics from a per-document topic distribution, and then words are sampled from a topic-specific word distribution. This entire process, including the dependencies between the observed and latent variables, is concisely represented using **plate notation**, as presented in Figure 2.10.

The learning process is fundamentally driven by Bayesian inference, specifically through iterative updates employing Bayes' theorem. This allows the model to refine its estimates of document-topic and topic-word distributions as it processes the data. Under the assumption that the document is a single test case or software delta (such as a list of changed file names), the model is described by several parameters: **W** denotes all words in all documents, **Z** identity of the topic of all words in all documents, θ is a distribution of topics in document d, and α and β describe the prior weights of topics in each document and the prior weight of words in each topic. Vectors **W** and **Z** are drawn from categorical distributions, and θ is from the Dirichlet distribution.

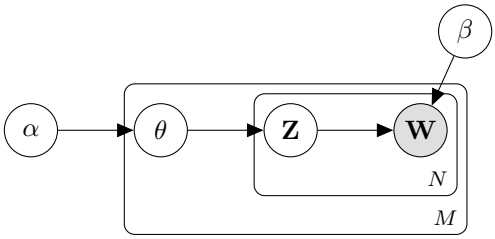


Figure 2.10: Plate notation for LDA

The plate notation diagram in Figure 2.10 provides a concise visual representation of the generative process LDA and its conditional dependencies. The plates signify repeated structures in the model. The outer plate, denoted by M, represents the collection of documents in the corpus. The inner plate, denoted

by N, represents the words within each document. The total probability of an LDA model is expressed as a joint probability distribution on all observed words ($\mathbf{W}$), latent topic assignments ($\mathbf{Z}$), and document-topic distributions (θ), conditioned on the Dirichlet prior parameters (α, β) and topic-word distributions (φ). Although not explicitly shown inside a node on the plate diagram, the topic-word distribution, φ, is a key latent parameter that governs the probability of a given word occurring in a given topic, and is a central part of the model's generative process. This joint probability decomposes according to the graphical structure of the model, highlighting the conditional dependencies, as shown in Equation (2.43).

$$\begin{aligned} p(\boldsymbol{W}, \boldsymbol{Z}, \boldsymbol{\theta}, \boldsymbol{\varphi}; \alpha, \beta) = & \left(\prod_{i=1}^{K} p(\varphi_i; \beta) \right) \\ & \times \left(\prod_{j=1}^{M} p(\theta_j; \alpha) \right) \\ & \times \left(\prod_{j=1}^{M} \prod_{t=1}^{N_j} PpP(W_{j,t} \mid \varphi_{Z_{j,t}}) \right) \end{aligned} \tag{2.43}$$

where:

- K denotes the total number of topics assumed in the model.
- M represents the total number of documents in the corpus.
- N_j is the number of words in document j.
- $\boldsymbol{W} = (W_{j,t})$ represents the observed words, where $W_{j,t}$ is the t-th word in document j.
- $\boldsymbol{Z} = (Z_{j,t})$ denotes the assignment of latent topics for each word $W_{j,t}$.
- $\boldsymbol{\theta} = (\theta_j)$ describes the topic distribution for each document j, where each θ_j is a probability vector over K topics, drawn from a Dirichlet distribution with parameter α.

- $\boldsymbol{\varphi} = (\varphi_i)$ represents the word distribution for each topic i, where each φ_i is a probability vector over the vocabulary, drawn from a Dirichlet distribution with parameter β.
- α and β are the Dirichlet prior parameters that influence the sparsity of topic distributions in documents and word distributions in topics, respectively.

It is important to note that LDA is an unsupervised method that requires a predefined number of topics, namely $K \in \mathbb{N}$ as a hyperparameter. The value of K is not arbitrarily chosen, but is typically determined through a systematic hyperparameter tuning process, with the optimal value depending on the size of the corpus and the desired granularity of the topics. The topic discovery process is entirely data-driven, without any prior knowledge of the document context or content meaning. Therefore, LDA is a "bag-of-words" approach that does not consider semantic relationships or other structural information within documents.

Using a previously defined phrase, `Interaction between 256QAM and DFT-s-OFDM for TDD PUSCH`, and assuming $K = 3$ and its presence in some document d, we can derive a possible and exemplary topic distribution for this document:

- **Topic 1**, possibly related to modulation:

  $p(T1|d) = 0.4$, with high-probability words: 256QAM, modulation, amplitude
- **Topic 2**, possibly related to waveforming:

  $p(T2|d) = 0.35$, with high-probability words: DFT, OFDM, transform
- **Topic 3**, possibly related to the channel:

  $p(T3|d) = 0.25$, with high-probability words: PUSCH, uplink, TDD

Therefore, extending this concept, diverse forms of textual data, such as system logs or software delta commit messages, can be transformed into probabilistic vectors. Based on this, Section 4.2 details how LDA-encoded data integration with a specifically designed classifier forms the core of the LMLDA. Its key innovation is a novel combination of outputs from two distinct LDA applications: one

describing thematic structures in test-related datasets, and another identifying themes within software deltas. These two independent probabilistic representations are then fused into a joint distribution which then serves as the rich input for the classifier.

3 Literature Review

Noticeably, the problem of automated anomaly detection (test set optimization or ticket assignment) is not new and has been addressed for many years. Luca et al. [36] and Cubrani et al. [32] pursued the idea of automating such a process. Anvik et al. [6] tried to answer the question "Who should fix this bug?". These ideas were followed in further work by Jonsson et al. [61] or Alreshedy et al. [4]. It was noted that bug reports often contain logs that have crash dumps. Then, Kallis et al. [63] proposed to take advantage of machine learning strategies in ticket titles and descriptions to automatically label GitHub issues. Similar studies have been performed in industrial applications, where such a scenario has several additional demands. Sarkar et al. [106] have made an effort to adopt an automated text classification pipeline to reduce bug-fixing costs in Ericsson. Thus, both academic and industrial research increasingly focuses on applying machine learning to automated anomaly detection in software engineering. This chapter focuses on a review of the literature related to this field and is divided into three sections that cover key areas in applied machine learning: unsupervised models, supervised models, and probabilistic approaches that include uncertainty estimation. At the end of each section a table summarizing this dissertation's contributions to these specific research areas is given.

3.1 Unsupervised methods

In modern software testing, sophisticated clustering methods are used to reveal latent patterns and reduce redundancies in extensive test suites. This chapter examines how clustering algorithms are used in conjunction with distance metrics to assemble test cases based on similarity measures, thereby enabling advanced test case prioritization and suite reduction strategies.

3.1.1 Clustering

Hierarchical Agglomerative Clustering: Yoo et al. [139] used run-time information from test execution traces, such as instructions and their execution counts. Their work, which used a mixed methodological approach for a one-time evaluation, also considered the robustness of the model to noisy data and incorporated manual supervision along with classical benchmarks for comparison, thus strengthening the credibility of the clustering results. Mahdieh et al. [82] used coverage information, calculating similarity using Euclidean, Manhattan, and cosine distances. The rationale behind using code coverage is that diverse coverage is more likely to reveal different faults. In addition, they addressed issues related to data imbalance using a thorough traditional benchmarking scenario for comprehensive analysis. Carlson et al. [22] used code coverage similarity, code complexity (using lines of code and method dependency count), and fault history. They then prioritized test cases within each cluster using these same metrics and combined the prioritized clusters using a round-robin approach. It should be noted that, while the round-robin technique ensures a fair cyclic selection process, it does not explicitly weigh the importance or effectiveness of individual tests. Zhao et al. [145] introduced a hybrid test case prioritization method combining code coverage similarities with Bayesian networks. The test cases are first clustered using agglomerative hierarchical clustering based on code coverage and then prioritized on the probability of failure within the clusters using a Bayesian network model that incorporates code changes, coverage data, and software quality metrics. This hybrid method, which also employs classical benchmarks in its one-time evaluation, demonstrates an effort to blend execution data with statistical inference for better prioritization. Coviello et al. [31] pro-

posed a method in which similar test cases are clustered according to coverage and then reduced by selecting the most representative test case, namely the one covering the most statements, of each group. Finally, Viggiato et al. [126, 127] proposed a method to identify similar test cases by clustering similar test steps written in natural language. The team preprocessed test cases, clustered steps using various word / sentence embeddings, similarity metrics, and clustering algorithms, and subsequently identified similar test cases based on these clustered steps. Their evaluation, conducted on a large set of test cases and steps from a game project, compared different technique combinations, thereby offering insights into the benefits of using textual data alongside execution characteristics. In this work, hierarchical clustering served as a primary tool, with an additional focus on alternative strategies such as K-means.

K-Means: Shimari et al. [111] used it for the selection of test cases based on the run-time information of test cases encoded in vectors of integers, with Euclidean distance measuring vector similarity. This unsupervised method was applied in an industrial one-time evaluation scenario, although it did not explicitly address data imbalance or model robustness. Yu et al. [140] grouped test cases based on features from log data, including log template counts, order, and semantics, reflecting a potential bias awareness in their industrial context. Chetouane et al. [28] combined K-means clustering with binary search to reduce the test suite. Their mixed methodology, also evaluated in an industrial setting, was attentive to potential biases, although it did not focus on data imbalance. Zhang et al. [143] used K-means clustering on function execution profiles of modification-traversing test cases as input to their regression test selection technique. In particular, while their approach initially operated without continuous supervision, manual intervention was later incorporated to fine-tune the results. Beyond fully unsupervised methods, Chen et al. [26] introduced a semi-supervised approach by creating a map of two constraints–must-link and cannot-link–to initialize the clustering process. This additional layer of constraint mapping marked a departure from typical fully unsupervised methodologies in clustering test cases. Arafeen et al. [7] introduced an approach that evaluates the textual similarity between requirements and tests, to assess the significance of words within documents. Furthermore, the industrial applicability of test clustering was demonstrated by

another study [75] that employed a two-step approach: first using hierarchical clustering to initialize K-means for post-processing to optimize both the size and the number of clusters in a mobile application context.

K-Medoids: Chen [24] first calculates the Manhattan distance between test cases (treated as strings) and then clusters them using an improved K-medoids algorithm featuring a Fermat point-inspired center selection. Within each cluster, test cases are ranked using a min-max greedy strategy. In this approach, the Min-Max greedy algorithm iteratively selects the option with the best worst-case outcome, ensuring resilience against worst-case scenarios, a detail that underscores the robustness of the clustering strategy. Liu et al. [76] aimed to reduce test suites, using the coverage and complexity of test cases as input features, with similarity calculated using the Euclidean distance. They subsequently removed test cases with the lowest coverage within each cluster. Pei et al. [95] investigated clustering methods – including K-means, K-medoids, and hierarchical agglomerative clustering – to organize test source codes by their text, deducing that closer test cases likely share similar fault detection capabilities.

3.1.2 Topic Modeling

Researchers often classify tests or code fragments based on a common vocabulary. Recognizing that numerical metrics do not capture the nuanced code-test case relationship, they have turned to textual analysis of code and tests using NLP methods. A review by Silva et al. [112] revealed that LDA is the most widely used and well-understood topic modeling technique, primarily for processing bug reports. LDA is a probabilistic generative model that is used to represent documents as combinations of hidden topics (groups of words) [16]. For example, Thomas et al. [120] used LDA for static black-box test case prioritization by modeling the linguistic data within test cases (identifier names, comments, string literals) to approximate functionality and prioritize diverse testing areas, with the aim of earlier fault detection. Lachmann et al. [69], introduced a straightforward approach to prioritize system-level test cases by transforming natural language test descriptions into word frequency vectors, allowing for the identification of latent patterns. Building on this, Chen et al. [27] empirically found that topics represented with fewer tests and higher defect rates require more testing effort,

suggesting that topic models can identify undertested, defect-prone areas by generating high-level system views from source and test file linguistic data. Further extending LDA's application, Xia et al. [134] proposed a framework for automated bug triaging, where they incorporated product and component information into topic generation to map words in bug reports to relevant topics, improving developer recommendations. In a similar study, Le et al. [70] used LDA to encode data for classification, predicting the effectiveness of bug localization tools by integrating suspiciousness scores, textual content, topic model distributions of bug reports, and metadata.

3.1.3 Summary

Table 3.1 provides an overview of unsupervised and mixed-method approaches in software engineering, focusing primarily on clustering and topic modeling for TSO. These listed approaches typically perform one-time evaluations and often do not explicitly handle data imbalance or robustness to noise. This work typically focuses on a single data aspect, mainly the testing set. However, Chapter 4 of this dissertation offers a distinctive contribution to unsupervised models with a delta-centric clustering method, departing from conventional test-focused evaluations. This arises from noticing the limitations of previously proposed techniques, which improperly fragment functional test groups and incorrectly assume uniform test dynamics and significance. Chapter 4 later argues that linking code changes to bugs is not a straightforward causation and that current methods mishandle architectural groupings. It applies delta-based clustering to expose evolutionary patterns and hidden dependencies by examining software delta sequences, improving test selection by maintaining architectural group awareness, and ensuring thorough coverage. Furthermore, Chapter 4 introduces LMLDA, asserting that effective test suite optimization requires understanding both past test performance and the nature of code changes, balancing coverage with reduced computational effort. Although the literature often isolates test selection from software delta analysis, this chapter stresses their crucial overlap where anomalies appear.

Table 3.1: Summary of research characteristics for unsupervised methods

Reference	Data type	Handles imbalanced data	Industrial scenario	Extended metrics	Evaluation scenario	Method or algorithm	Manual supervision	Classical benchmarks	Robustness to noise	Discusses bias
[139]	SW			x	OT	Mix	x	x	x	x
[82]	SW	x			RW	Mix		x		
[22]	SW		x		OT	Unsup		x		x
[145]	SW				OT	Mix		x		
[31]	SW				OT	Mix		x		
[126], [127]	Test		x	x	OT	Mix		x		
[111]	SW		x		OT	Unsup		x		
[140]	SW		x		OT	Mix		x		x
[28]	SW		x		OT	Unsup		x		x
[143]	SW				OT	Unsup	x	x		
[26]	SW				OT	Mix	x	x		
[7]	Test				OT	Mix		x		
[75]	Test		x		OT	Unsup				
[24]	Test			x	OT	Mix		x	x	
[76]	SW				OT	Mix		x		
[95]	SW				OT	Mix		x		
[120]	Both			x	OT	Unsup				
[69]	SW		x		OT	Mix				
[27]	Both			x	OT	Unsup				x
[134]	SW		x	x	OT	Unsup				
[70]	SW			x	OT	Mix				
[115]	Test			x	OT	Mix		x	x	x
Chapter 4	Both	x	x	x	RW	Mix	x	x	x	x

Abbreviations:
SW: Software-related, Test: Test-related
OT: One-time, RW: Rolling-window
Mix: Mixed, Unsup: Unsupervised

3.2 Supervised and deep learning methods

Supervised learning models are widely applied in software engineering to tackle critical tasks such as anomaly classification and TSO. These techniques, using labeled historical data, allow systems to learn complex patterns and make predictions. This section presents a detailed examination of supervised and deep learning methods, with particular attention to their application in the refinement of test sets and precisely the classification of software bugs.

3.2.1 Classical approaches

Starting from basic approaches, Palma et al. [92] applied logistic regression to analyze and rank test cases based not only on traditional numerical metrics but also on similarity-based measures (for example, the Hamming distance between data points). In parallel, Dejaeger et al. [35] adopted a naive Bayes classifier to categorize test cases by quantifying altered code lines and branch executions, underscoring the need to extend datasets beyond simple numerical features.

Support Vector Machine (SVM): function by identifying an optimal hyperplane that maximally separates data points of different classes in a high-dimensional space. This hyperplane is defined by support vectors, the closest points to the decision boundary, establishing the margin between classes. The objective is to maximize this margin to improve generalization on new data. Choudhary et al. [29] designed a SVM to assign priorities to Firefox crash reports by focusing on features such as crash frequency and entropy. Busjaeger et al. [21] developed an SVM-based binary classifier to order test cases from historical data in an industrial setting, while Grano et al. [48] used SVMs to assess branch coverage, prioritizing tests in components predicted to have high coverage and reallocating resources to critical areas with low predicted coverage. Lachmann et al. [69] further demonstrated that preprocessing natural language test descriptions to create a rich vocabulary can significantly enrich the feature space, increasing the accuracy of SVM-based classifiers. However, as reported, SVMs tend to struggle in scenarios characterized by large, imbalanced datasets, where the model can become biased towards the majority class, leading to suboptimal performance in the minority class.

Tree-based methods: Alenezi and Banitaan [3] applied it to prioritize bug reports by comparing feature sets derived from word-frequency weighted words versus classified bug report attributes. Their work concluded that the latter achieved superior performance, as the word-frequency approach often overlooks the true semantic meaning of text. Similarly, Bertolino et al. [14] compared several models and highlighted that tree-based boosting algorithms can provide high precision in regression testing tasks. Lenz et al. [71] adopted a two-step approach in which the test data was first grouped and then classified using the C4.5 decision tree algorithm [98], which selects optimal splits based on information entropy. Chen et al. [25] evaluated the XGBoost framework on both artificially generated datasets (with mutation faults) and industrial datasets with authentic faults, showing that its performance critically depends on the nature of the input data – proving that industrial data are much more challenging. However, despite all positives, decision tree-based methods are also well known for their instability and propensity to overfit, often requiring significant maintenance in the form of tree pruning and periodic retraining, especially in dynamic environments.

3.2.2 Neural networks

Lousada et al. [79] introduced a shallow neural network that computes the dot product of the file and test embeddings followed by a dense prediction layer. Sharif et al. [110] advanced this idea with DeepOrder, a deep neural network that learns a prioritization function from historical CI data while explicitly managing the imbalance of failed tests. Similarly, Mahdieh et al. [81] implemented a simple two-layer network in which coverage metrics were linked with fault detection probabilities to weight components by fault frequency, and Marijan et al. [84] deployed a 6-layer NN trained on both code changes and test features to predict fault detection ability in regression testing scenarios. Recurrent architectures have also been explored: Xiao et al. [135] and Saidani et al. [104] applied LSTM networks to predict the probability of test case failures and CI build results, respectively. However, Baoxin Wang [128] pointed out that LSTM networks struggle to extract position-invariant features – an essential capability when processing natural language descriptions of test cases. This limitation motivated the adoption of alternative approaches, such as Convolution Neural

Network (CNN). Umer et al. [123] used a CNN to predict the priority of the bug report, which captures local semantic patterns but may lose global context. More recently, Samoa et al. [105] and Phan et al. [97] used Graph Neural Networks (GNN) in flow-augmented abstract syntax trees to predict test execution time and detect defects. These GNN-based approaches underscore the importance of structural representation in code; however, their reliance on large, well-labeled datasets and their intrinsic "black-box" nature limit their interpretability and ease of integration into evolving test and anomaly detection processes.

LLM: in a survey of relevant studies, Wang et al. [129] found that the most common applications are test case preparation and program repair, with prompt engineering strategies dominated by zero-shot and few-shot learning. Schäfer et al. [107] provide an effective example of a zero-shot approach with their method, which generates unit tests. The tool's adaptive prompting technique provides the LLM with a function's signature, source code, and documentation examples, and if a test fails, it iteratively re-prompts the model with the failing test and error message to guide it toward a fix, achieving significantly higher code coverage than prior state-of-the-art tools. For Automated Program Repair (APR), Xia et al. [133] conducted an extensive study in multiple LLMs and repair settings, demonstrating that these models can outperform specialized APR tools without relying on bug fix datasets. Their research demonstrated that supplying suffix code in a fill-in-the-blank context is essential for producing accurate fixes in a code. However, the generative nature of LLMs can result in unreliable test cases with low fault detection capability, which is the root of the desired anomaly detection pipeline. To address this, Dakhel et al. [33] introduced a method that iteratively augments prompts with a "surviving mutants", namely faults that initial tests failed to detect, consequently guiding the LLM to generate more effective, bug-revealing tests by focusing on weaknesses in the existing test suite. Despite these advances, significant challenges remain. LLMs impose substantial computational and energy requirements, a point underscored by Jiang et al. [59], who evaluated multiple models and created a new benchmark to prevent data leakage. They found that while fine-tuning can improve bug-fixing performance by up to 2%, even standard models fixed 72% more bugs than specialized deep-learning APR techniques, highlighting both the power and the cost. Data quality can also

severely undermine model performance. Gong et al. [47] investigated class overlap problem, where defective and non-defective modules are metrically similar, and found that it harms not only model performance but also the stability of feature-importance rankings. Likewise, Tu et al. [121] showed that disregarding the chronological sequence in issue tracking systems for attributes such as summary, component, and assignee exposes "future" information, resulting in falsely positive assessments. Finally, persistent issues with the frequency of successful test cases in historical data and intellectual property rights reduce the practical effectiveness of LLM-based solutions in industrial uses [5].

3.2.3 Summary

Table 3.2 summarizes key characteristics of supervised learning models applied in software engineering, mainly for the classification of anomalies, the prioritization of bugs, and the prioritization of test cases. As with unsupervised methods, most existing supervised approaches typically focus on one-time evaluations. Although some address data imbalance and certain LLM applications operate in industrial settings with extended metrics, consistent integration of dynamic evaluation, comprehensive handling of imbalances, and human-in-the-loop feedback remains less common. Unlike the previously mentioned works, the dissertation, particularly in Chapters 4 and 5, distinguishes itself by implementing a dynamic rolling-window evaluation strategy that utilizes historical software change data. In Chapter 4, a new machine learning model, the LMLDA, is introduced. This model uses a unique form of logistic regression, needing only a few neurons to effectively learn the joint probability of contextual co-occurrences in tests and software changes. This capability facilitates strong predictive outcomes without relying on expensive LLM. These efforts mitigate the limitations often seen in traditional supervised models, leading to a cost-efficient, reliable and context-aware prioritization of test cases. On the other hand, Chapter 5 examines the targeted use of LLMs within an industrial setting, focusing on techniques to improve their contextual comprehension by fine-tuning them for specific industrial datasets. It contrasts prompt engineering with the requirement for local hosting to gain greater and more detailed control over model functionalities. Furthermore, Chapter 5 focuses on the inclusion of a Bayesian methodology in these models, which

allows for a detailed quantification of uncertainties. This innovation makes it possible to precisely evaluate the model's confidence in its predictions and the dependability of user-supplied data, thus creating an essential human-in-the-loop system that boosts trust and interpretability in real-world software engineering scenarios.

Table 3.2: Summary of research characteristics for supervised methods

Reference	Data type	Handles imbalanced data	Industrial scenario	Extended metrics	Evaluation ecenario	Method or algorithm	Manual supervision	Classical benchmarks	Robustness to noise	Discusses bias
[92]	SW				OT	Sup		x		
[35]	SW			x	OT	Sup		x		
[29]	SW				OT	Sup				
[21]	SW		x		OT	Sup		x		
[48]	SW				OT	Sup				
[3]	SW				OT	Sup		x		
[14], [84]	SW				OT	Sup		x		
[71]	SW				OT	Sup		x		
[25]	SW		x		OT	Sup				
[79]	SW			x	OT	Sup		x		
[110]	SW	x		x	OT	Sup		x		
[104]	SW				OT	Sup		x		
[81]	SW			x	OT	Sup				
[135]	SW				OT	Sup				
[123], [105], [97]	SW				OT	Sup				
[107]	Both				OT	Sup				
[133]	SW				OT	Sup		x		
[33]	SW			x	OT	Sup				
[59]	SW		x	x	OT	Sup		x		
[47]	SW	x		x	OT	Mix				x
[94], [60]	SW		x	x	OT	Sup	x	x		
[86]	SW	x			OT	Mix		x		
[122], [57]	SW	x	x	x	RW	Mix	x	x		x
Chapter 4	Both	x	x	x	RW	Mix	x	x	x	x
Chapter 5	Both	x	x	x	RW	Sup	x	x	x	x

Abbreviations:
SW: Software-related
OT: One-time, RW: Rolling-window
Mix: Mixed, Unsup: Unsupervised, Sup: Supervised

3.3 Probabilistic methods and uncertainty estimation

The adoption of automated systems hinges on trust, which requires not only precise models but also trustworthy confidence levels. This part explores existing studies on the assessment and mitigation of uncertainty within the predictions ML, contrasting them with the approach of this dissertation, with particular attention to LDA and the Bayesian framework that underlies this thesis.

3.3.1 Sources of uncertainty in software engineering

Model prediction uncertainty arises from a variety of origins and is typically classified into two essential categories: aleatoric uncertainty and epistemic uncertainty. The significance of differentiating these types of uncertainty in machine learning, along with a comprehensive overview of strategies to manage them, has been effectively explained and presented by Hüllermeier et al. [57]. Ulmer et al. [122] performed research on the impact of data scarcity and discovered that the total uncertainty of a model is influenced more by its uncertainty of the data (epistemic) than by the uncertainty of the model (aleatoric). In software engineering, key sources include skewed sample distributions. Here, uneven error distribution (for example, between departments) can introduce classifier bias. Osband et al. [91]) research extensively investigated aleatoric and epistemic uncertainty in machine learning, particularly finding that the method and data scarcity impact predictive uncertainty and calibration in NLP tasks. Incomplete context understanding from an evolving industrial corpus is another source, as identified by Cavalcanti et al. [23].

For language models, a primary measure of how well a model has adapted to a specific domain is **perplexity**. Language models are trained to predict the next token in a sequence by minimizing cross-entropy loss. The perplexity, since defined as the exponentiation of this loss, represents the uncertainty of the model in its predictions, as introduced by Jelinek et al. [58]. A lower perplexity score indicates that the model is less "surprised" by the text in the evaluation corpus. Perplexity scores are only comparable on the same corpus as described by Miaschi et al. [88], thus the goal is to observe a reduction in perplexity after fine-tuning a base model on a specific domain. Meister et al. [87] introduced a different

method to assess how language models learn natural language, focusing on evaluating statistical patterns and neural models. Then, Miaschi et al. [88] presented a linguistic investigation into how sentence structure affects the perplexity of LLMs.

3.3.2 Probabilistic approaches

Evaluating the confidence of an LLM's predictions is essential due to its lack of direct data knowledge. Several methods have been developed to approximate model uncertainty. Gal [43] demonstrated that the application of stochastic regularization methods such as dropout in standard neural networks mirrors the process of variational inference found in Bayesian networks. This study focuses on examining uncertainty in deep learning across diverse data types and testing situations such as active and reinforcement learning, employing Bayesian Deep Learning with a human-in-the-loop approach. In further work, Gal et al. [45] presented Monte Carlo Dropout (MCD), which involves repeatedly applying dropout during inference to a sample from a variational distribution, thereby approximating the predictive distribution through Monte Carlo integration. Fomicheva et al. [41] introduced general uncertainty quantification methods, focusing on an unsupervised technique for quality estimation in neural machine translation with MCD. The study distinguishes aleatoric from epistemic uncertainty, evaluates uncertainty, and compares uncertainty handling across models, proposing a human-in-the-loop framework. Glushkova et al. [46] combined MCD and deep ensembles to derive quality scores and confidence intervals in their uncertainty-aware evaluation. In a different approach, Li et al. [74] proposed a framework with the ability to capture the full effects of both aleatoric and epistemic uncertainty. More recently, Lorenz Kuhn [68] presented the novel idea of semantic entropy, which gathers batches that share the same semantics in clusters and then performs cluster-wise predictive entropy. Then, Talman et al. [117] introduced Bayesian uncertainty modeling, using a stochastic weight-average Gaussian technique to quantify uncertainty in natural language understanding tasks.

3.3.3 Bayesian deep learning

Xiao et al. [136] performed a large-scale empirical analysis of uncertainty quantification with pre-trained language models for various NLP tasks, focusing on how the model configuration affects overall uncertainty. Moreover, Vazhentsev et al. [125] focused on estimating the uncertainty of transformer predictions for misclassification detection in NLP tasks (e.g. text classification). Considering the potential of LLMs within a variational framework, research by Liu et al. [77] has shown the effectiveness of combining a BERT encoder with a dense variational layer in reducing classification errors. Furthermore, Vazhentsev et al. [125] focused on estimating uncertainty in transformer models to detect text misclassification. Building upon these prior ideas, the approach in this dissertation aims to utilize a LLM integrated within a Bayesian model to leverage its extensive contextual knowledge for the classification of highly specific software artifacts (bug descriptions) in the anomaly detection efforts. This leads to the novel concept of using an LLM encapsulated in a Bayesian model. This approach is expected to provide not only accurate classifications, but also valuable insight into the uncertainty associated with these predictions, contributing to a more robust and reliable TSO process. It is essential to build trustworthy tools that developers will not ignore as they tend to do so, as reported by Orso et al. [94] or Jonsson et al. [60].

3.3.4 Summary

Table 3.3 reveals a gap in the way software engineering research deals with uncertainty, specifically in differentiating types of uncertainty and incorporating human input consistently. Chapter 5 responds to this gap by presenting an innovative approach to assess uncertainty in the detection of LLM-based software anomalies, particularly with evolving and unbalanced data sets. It clearly differentiates aleatoric from epistemic uncertainties, pinpointing specialized terminology and abbreviations as significant, yet frequently overlooked, sources of epistemic uncertainty in industry settings. A key contribution of this work is the application of perplexity to measure LLM suitability and confidence in specialized texts, such as those of the corpus 3GPP, directly correlating this measurement

with classification accuracy. In addition, Chapter 5 examines how environmental factors affect LLM performance and devises a strategy to incorporate and elaborate abbreviations through a 3GPP-related dictionary, thus reducing contextual epistemic uncertainty. It affirms the effectiveness of fine-tuning LLMs to adapt to new token distributions and datasets. Ultimately, the chapter introduces a sophisticated human-in-the-loop system in which continuous human oversight and intervention substantially mitigate epistemic uncertainty, thus enhancing the practical utility and reliability of the whole solution.

Table 3.3: Summary of research characteristics for probabilistic methods

Reference	Differentiates uncertainty type	Estimates uncertainty	Compares uncertainty handling	Human-in-Loop to address uncertainty	Data imbalance as uncertainty source	Compute uncertainty for new data	Label smoothing for class imbalance
[58]		x	x			x	
[60]		x	x	x		x	
[122]	x	x	x	x	x	x	x
[57]	x	x	x	x	x	x	x
[91]	x	x	x	x	x	x	x
[88]		x	x			x	
[43]	x	x	x	x		x	
[45]		x	x	x		x	
[41]	x	x	x	x		x	
[46]	x	x	x	x	x	x	
[68]		x				x	
[117]		x	x			x	
[136]		x	x		x	x	
[125]		x	x			x	
[77]					x		
Chapter 5	x	x	x	x	x	x	x

4 Application of test set optimization (TSO) for anomaly detection

This chapter addresses the **efficiency** and **reliability** components of the dissertation thesis. It introduces a novel ML framework, LMLDA, designed to optimize the testing process within the CI/CD pipeline. By moving beyond simple quantitative metrics to a semantic understanding of code changes and tests, this framework aims to create a more efficient testing cycle by prioritizing the most critical tests, thereby accelerating anomaly detection without compromising the reliability that comes from comprehensive test coverage.

TSO methods typically rely on only one viewpoint: using either the content of the tests or their results. This narrow approach neglects the direct link between the deltas in the software components and their real effects on the testing results. Since software components undergo development on a daily basis, multiple deltas are directly affected by each other, potentially resulting in correlated test failures. Furthermore, the high rate of passing tests skews the data toward positive results, complicating the use of supervised learning. Therefore, unsupervised learning methods are frequently used due to their less complicated tuning requirements. However, clustering tests usually break the architectural composition of the original set. For example, functional test sets can unintentionally become fragmented into separate groups through clustering, risking incomplete architectural coverage. Another common flaw is the assumption that tests and code modifications (deltas) are uniformly dynamic and that all tests are of equal significance. Although clustering sorts tests based on similarity, it frequently

overlooks the nuances of individual critical test cases. For true optimization, it is essential to understand the direct interaction between a specific test and a particular software delta. Crucially, correlation does not imply causation; more lines of code changed do not necessarily mean more bugs. Test-focused clustering also struggles with existing architectural groupings. The potential advantage lies in delta-based clustering, which can uncover evolutionary trends and patterns within software. Analyzing sequences of deltas provides insight into how software adapts, which is vital for long-term maintainability and architectural evolution, aspects often obscured by static test metrics.

In Section 4.1, an initial clustering baseline is proposed. This two-step classifier clusters software deltas based on the magnitude of code changes, assuming that similar deltas correlate with similar test failures. This approach serves as a guide for further research and is designed for rapid early-stage anomaly detection. Subsequently, Section 4.2 introduces a new classifier built from scratch. It uses logistic regression to learn the language context of software deltas and tests, assessing the joint probability of high-risk points. The core objective of both methodologies is to avoid large and complex models like LLMs. Instead, we create smaller, task-oriented models that are easy to learn, maintain, and retrain frequently, ensuring that they remain effective for the constantly evolving telecommunications domain.

4.1 A two-step unsupervised test selection

The baseline solution shifts from traditional test-centric approaches to a **delta-centric** clustering strategy. To illustrate this process in practice, consider the following scenario:

1. **Input:** A new code change (delta) is introduced to the system, primarily modifying C++ files within a signal processing component.

2. **Processing:** The system searches historical data and finds a cluster of deltas that also previously affected this component and its C++ files. It then identifies that for these historical changes, the tests

`test_signal_processing_A` and `test_parameter_validation_B` failed most frequently.

3. **Output:** The system recommends executing `test_signal_processing_A` and `test_parameter_validation_B` as high-priority tests for the new delta, as there is a high probability that they will detect potential bugs.

The scientific basis for this shift lies in the consistent architectural impacts observed within clusters of deltas. The assumption is that modifications in comparable areas of the codebase frequently lead to similar failure patterns. This approach enables early anomaly detection using past failure data and is visually represented in Figure 4.1, which illustrates the flow proposed for test prediction.

- **New software delta**: The procedure starts with the introduction of a new software delta that represents a recent series of code modifications. It serves as the foundation for predictive analysis, with the objective of determining which existing tests are most likely to uncover problems associated with these modifications.
- **Clustering**: This action is performed to find similar deltas. The new software delta is analyzed and compared against a repository of past code changes. By grouping deltas that demonstrate similar characteristics or impact similar architectural components, past experiences are used to create an optimized testing set.
- **Extract failed tests**: Once similar historical deltas are identified, the system proceeds to extract the tests that failed for these deltas.
- **Predicted failing tests**: The extracted data is then used to generate predicted failure tests for the new delta. Failed tests for similar deltas are marked as high priority for execution.

Even though implementation faces constraints – historical data often show individual test case failures rather than entire test suite failures, which can lead to less accurate predictions – this research provides a framework for developing an advanced anomaly detection system within regression testing environments. The primary focus here is on designing a baseline prediction model that optimizes test

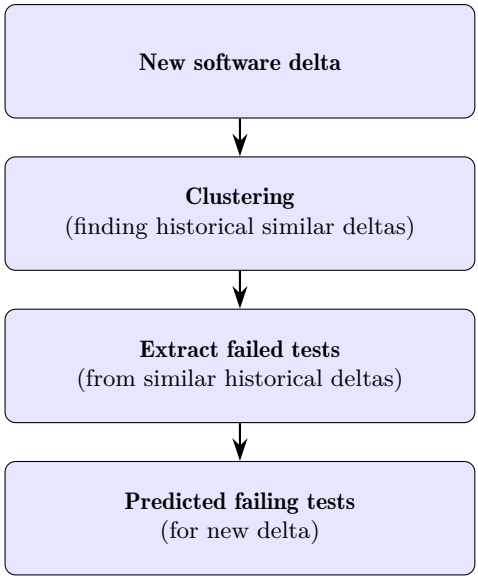


Figure 4.1: Delta-centric test prediction flow

case selection through the systematic evaluation of software deltas (incremental code changes).

Therefore, the driving goal of this section is to answer the research question **RQ 1**: To what extent can a purely quantitative delta-centric clustering approach effectively select failing tests and what are its main limitations in an industrial CI/CD environment?

4.1.1 Methodology

For two consecutive versions of deltas $\mathbf{d}_i$ and $\mathbf{d}_{i+1}$, we analyze the changes between the components and file types. Let:

- $\mathbf{C}_s = \{c_1, c_2, \ldots, c_N\}$ be the set of software components. The specific number of components, $N = 250$, is a fixed hyperparameter derived from the system's architecture, not an arbitrary assumption;
- define the set of supported file types as $\mathbf{F}_t$ = { cpp, py, yaml, txt, c, h,

hpp, java, pdl, pem, config, sh, bat, bb, bbclass, ims2, xml, env, cc, hh, lua, other };

- $\Delta_c = \{\text{additions}, \text{deletions}\}$ be the set of change types.

Then, the total number of features n in the vector $\mathbf{x}$ is $n = |\mathbf{C}_s| \times |\mathbf{F}_t| \times |\Delta_c| = 250 \times 22 \times 2 = 11{,}000$.

For each delta between versions d_i and d_{i+1}, we define the feature vector $\mathbf{X}$ as:

$$\mathbf{X} = [x_{c,f,d}]_{c \in \mathbf{C}_s, f \in \mathbf{F}_t, \delta \in \Delta_c} \tag{4.1}$$

where $x_{c,f,d}$ represents the number of lines changed (either added or deleted) for component c, file type f, and change type Δ.

The Algorithm 1 describes the process for generating the feature vector used in a delta-centric test prediction model. The procedure iterates through all data points to create a set of features. Each feature represents the number of code changes for a specific delta within a particular functional area of a given component. This structured feature generation is crucial for translating raw software changes into a format suitable for subsequent machine learning classification steps.

Algorithm 1 Generation of features for delta-centric test prediction

```
X ← ∅
for all c ∈ C_s do
  for all f ∈ F_t do
    for all δ ∈ Δ_c do
      feature_name ← concatenate(c, f, δ)
      feature_value ← count_changes(c, f, δ)
      X ← X ∪ {(feature_name, feature_value)}
    end for
  end for
end for
return X
```

Features with zero variance values in the dataset produced by Algorithm 1 were removed. Information on software components that do not change or always

do so in the same way would not have benefited from the clustering. This has saved memory and additional time that would have been spent computing a larger data matrix than necessary. It should be noted that for this dataset about 25% of the columns related to changes in the components can be removed this way.

When selecting an appropriate clustering algorithm, the choice of distance metric plays a crucial role in determining the similarity between data points. Based on analysis of the prior literature in Chapter 2.4.1 (p.42) and Chapter 3.1 (p.70), three clustering methods are considered: **Hierarchical Agglomerative**, K**-Means**, and K**-Medoids**. These are defined primarily by the parameter of the target number of clusters. The mean and standard deviation for each software version were calculated based on the number of clusters and the clustering metrics are used: namely, DBI defined in Equation (2.5) and Silhouette score given in Equation (2.4). They assess how well the data points are grouped based on their similarity, which is the primary goal of the clustering phase. In contrast, APFD, TSR, and True Positives Rate (TPR) are covered later in the results section to show how clustering affects the changes in metrics in TSO.

4.1.2 Empirical findings

To evaluate the effectiveness of each of the three clustering methods (Hierarchical, K-Means, and K-Medoids), a series of experiments was carried out. The key hyperparameter was the target number of clusters (K). To test the effectiveness of each method, a range of values K was tested: from 2 to $\sqrt{N}$, where N is the number of software versions under analysis. This served as a practical cap to avoid excessive data fragmentation, ensuring that each cluster represents a significant group rather than a single anomaly.

Table 4.1 presents the aggregated results as the mean and standard deviation of all hyper-parameters used in a research. An analysis of the table reveals several key observations. First, the hierarchical and K-Means methods achieve better cluster quality (lower DBI, higher Silhouette) than K-Medoids. Second, despite these differences, all three approaches produce very similar values for the *APFD* and *TPR* metrics. Despite the variations in the clustering quality metrics (*DBI* and *Silhouette* scores), the consistency of the test selection evaluation metrics (*APFD* and *TPR*), indicates their similar ability to identify the main and most

Table 4.1: Aggregated metrics by each method (mean ± standard deviation)

	Hierarchical	K-Means	K-Medoids
DBI	1.74 ± 0.39	1.82 ± 0.27	2.86 ± 0.84
Silhouette	0.37 ± 0.06	0.39 ± 0.06	-0.08 ± 0.06
APFD	0.56 ± 0.05	0.56 ± 0.05	0.57 ± 0.05
TSR	0.32 ± 0.15	0.32 ± 0.15	0.42 ± 0.19
TPR	0.78 ± 0.13	0.78 ± 0.13	0.70 ± 0.18
Max cluster size	54.69 ± 11.12	43.56 ± 4.50	48.23 ± 16.82

frequent faults. The challenge persists in detecting subtler faults due to the inherent fault distribution of the software. Each clustering approach usually identifies 70% to 80% of potential problem areas, indicating "low-hanging fruit" in fault detection. Therefore, most faults are located where various test strategies can easily find them. This creates a broad view, where few areas cause the most frequent problems.

As shown in Figure 4.2, the correlation matrix of the clustering and evaluation metrics provides valuable information, merging scientific and engineering perspectives. There is a notable inverse correlation between *Silhouette* and *DBI* (-0.780), highlighting their contrasting roles in the evaluation of clusters. Their weak correlations with testing metrics (e.g., *Silhouette* with *TPR* at 0.253, *TSR* at 0.271) imply that geometric cluster quality alone is not sufficient for effective testing. This weak correlation provides insight into the direction in which testing metrics relate to the shape of clusters. This is understandable since *TPR* measures how well samples are classified within their true clusters (relating to *Silhouette*'s assessment of cluster distinctness), while *TSR* evaluates the overall testing effectiveness (connecting to *DBI*'s measure of inter-cluster separation). Then, the positive correlation observed between *APFD* and *TSR* (0.635) suggests that even with reduced test sets, fault detection can be maintained or improved. However, the strong negative correlation between *TPR* and *TSR* (-0.904) highlights a critical trade-off: significant reduction in tests can severely hinder fault detection. Thus, a careful balance is needed to optimize the size of the test suite without compromising fault detection, avoiding the risks of excessive reduction.

Figure 4.3 presents a broader view on the result of several experiments related

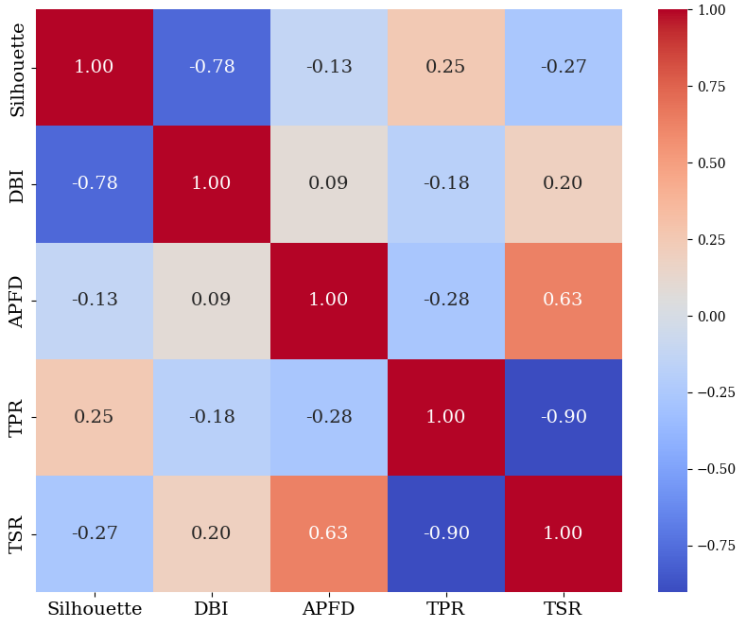
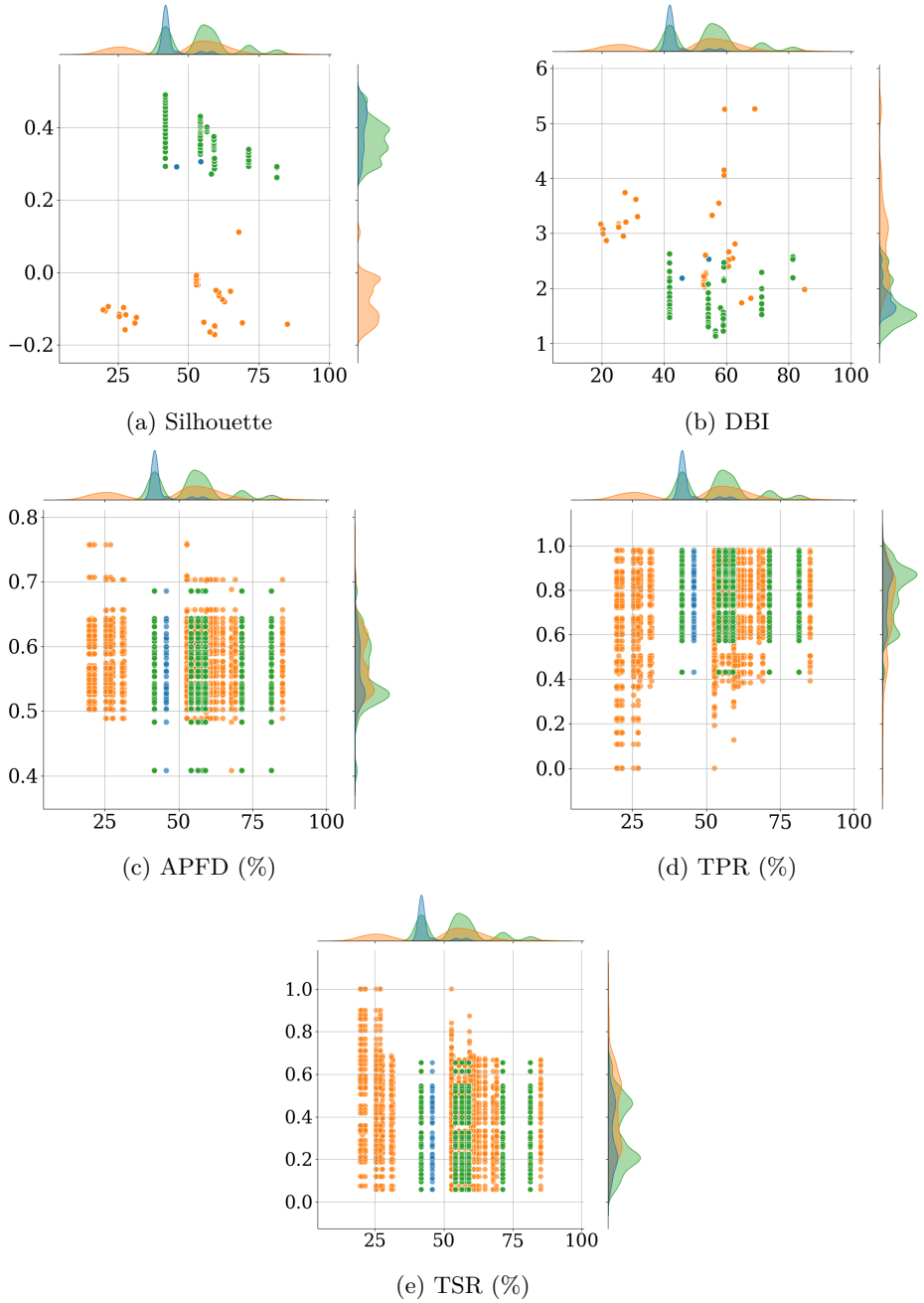


Figure 4.2: Correlation matrix of metrics related to optimization with delta-centric clustering



(a) Silhouette

(b) DBI

(c) APFD (%)

(d) TPR (%)

(e) TSR (%)

Figure 4.3: Density of clustering metrics. X-axis: max cluster size (%). Y-axis: metric density. Methods: **Hierarchical Agglomerative**, K-Means, K-Medoids

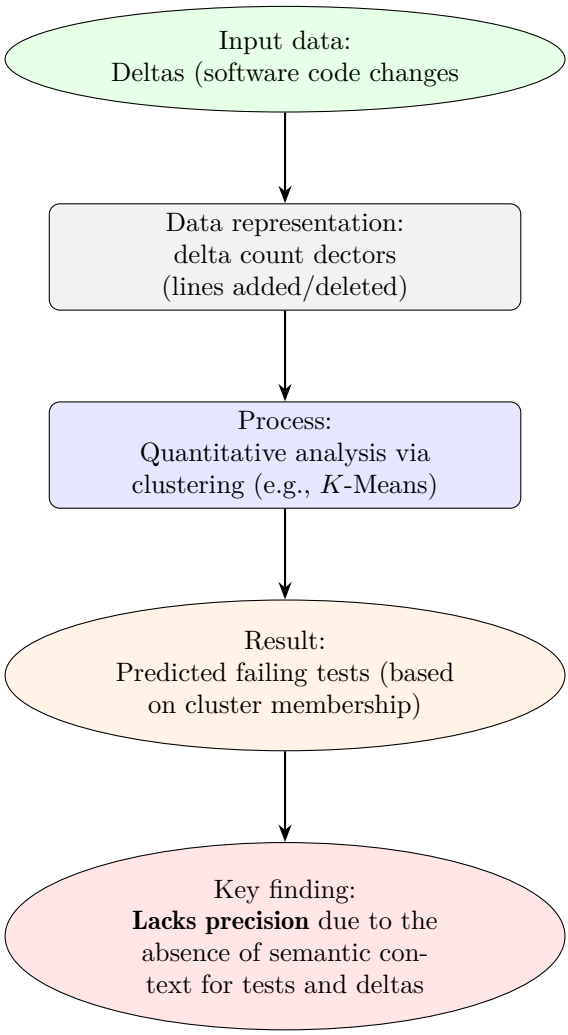
to the search for the best parameters. Although Hierarchical and K-Means offer a better balance of cluster quality and true positive rate, K-Means more often achieve a higher *TSR* with smaller clusters. The usual trade-off observed with *TPR* implies that the more aggressive reduction strategy, while efficient in terms of the size of the test set, could be to remove the tests crucial for revealing these rarer and more elusive faults. Additionally, mainly K-Means is able to work with smaller clusters, which does not directly translate to better overall results. This strongly implies that the more difficult challenge, as highlighted in the analysis, lies in locating the remaining subset of unpredicted tests. The subtle differences in the wrong classification patterns among the methods, despite their overall similarity of *APFD* and *TPR*, become critically important here. Conversely, the more conservative reduction of Hierarchical and K-Means, while offering slightly less test set reduction, preserves a higher proportion of true positives, suggesting that they might be better at retaining tests that target these harder-to-find defects.

4.1.3 Conclusion for unsupervised approach

The general flow of the entire method with the most profound key finding from the evaluation is presented in Figure 4.4. The problem of TSO in this context is not solely about maximizing fault detection on average, but specifically about minimizing false negatives for those hard-to-find faults while still achieving a meaningful *TSR*. The fact that each method makes slightly different mistakes opens opportunities for future work. Methods using clustering methods make similar mistakes – often in the same software versions but noticeably less often. This is due to the fact that the errors that occur at the time of prediction have not occurred before, neither in the closest history nor in the software versions in which the clustering methods were found to be similar to each other. This is one of the most important drawbacks of the presented solution, which has a very limited way to find unique and one-time bugs in the software. Therefore, a next step is to incorporate additional information into the process. As code changes were defined to be rather uniform in nature, the context of code commit (thus: description of changes and its relation to the content of tests that failed) or any other data that defines the context of the file seems to be a natural extension

of the dataset with additional and important content. That is, the knowledge related to the test contents and details on files that were changed by a recent commit (e.g. names of updated files). Thus, the approach would transfer the problem into a field of NLP in which much more context may be used. This next step is presented in Section 4.2.

Figure 4.4: Conceptual flow of the unsupervised baseline approach



4.2 Linear Model of Latent Dirichlet Allocation (LMLDA)

The previous section presented a baseline approach based on clustering, which grouped deltas according to the *magnitude* of code changes. While this method proved effective at identifying major faults, its key limitation, as highlighted in the conclusion, is the lack of analysis of the *semantic context* of the changes themselves. The system did not understand *what* the changes were about, only *how large* or *in which areas* they were.

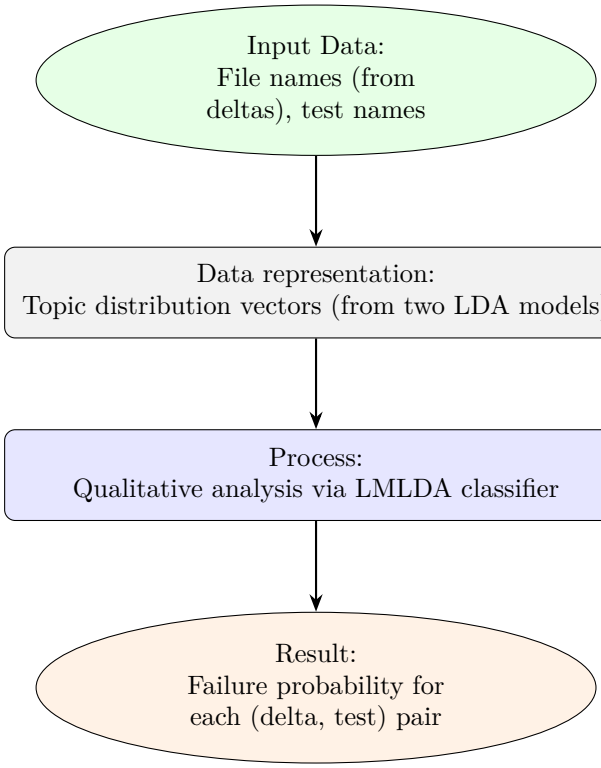
To address this limitation, this section introduces the LMLDA. It represents a significant extension that shifts from quantitative to qualitative analysis. Figure 4.5 provides a conceptual flow of the idea. Using NLP techniques, LMLDA analyzes the content (file and test names) to understand the thematic relationship between code modifications (deltas) and test cases. It is a generative classifier that models the joint probability distribution $p(\mathbf{x}, y)$ of features $\mathbf{x}$ and class labels y. This approach fundamentally differs from discriminative models like standard logistic regression, which only model conditional probability $p(y|\mathbf{x})$. This allows the model to predict failures not just based on structural similarity but primarily on contextual alignment, marking a step forward towards more precise anomaly detection.

A key advantage of this generative nature is the separation of model complexity between training and inference. During training, LMLDA captures non-linear relationships within the data, behaving similarly to a logistic regression model. However, for inference, its decision boundary is linear with respect to the input features, making predictions computationally efficient and equivalent to linear regression. This allows the model to learn complex patterns without sacrificing prediction speed.

An outline of the proposed methodology, along with sample data, is presented in Figure 4.6. The following paragraphs offer a step-by-step explanation of the approach, including tokenization, vectorization, LDA encoding, classification, and prediction using the proposed LMLDA model, along with the evaluation metrics used to assess the overall model performance.

Eventually, the research question that needs to be answered is **RQ 2**: Can a generative classifier such as LMLDA that models the semantic relationship be-

Figure 4.5: Conceptual flow of the LMLDA approach



tween the textual content of the code changes and the test names significantly improve test prioritization and accelerate defect discovery compared to quantitative baselines?

4.2.1 Tokenization

Tokenization involves segmenting a text into smaller elements known as tokens, which may be words or subwords. For example, as depicted in Figure 4.6, the filename “5GMaxRevision.cc” can be divided into tokens like “Max”, “Revision”, and “.cc”. Furthermore, application of a single tokenizer across corpora, trained both on files and tests, allows for consistent interpretation of both sets and enables more detailed text processing. Tokenization results in a sequence of integers,

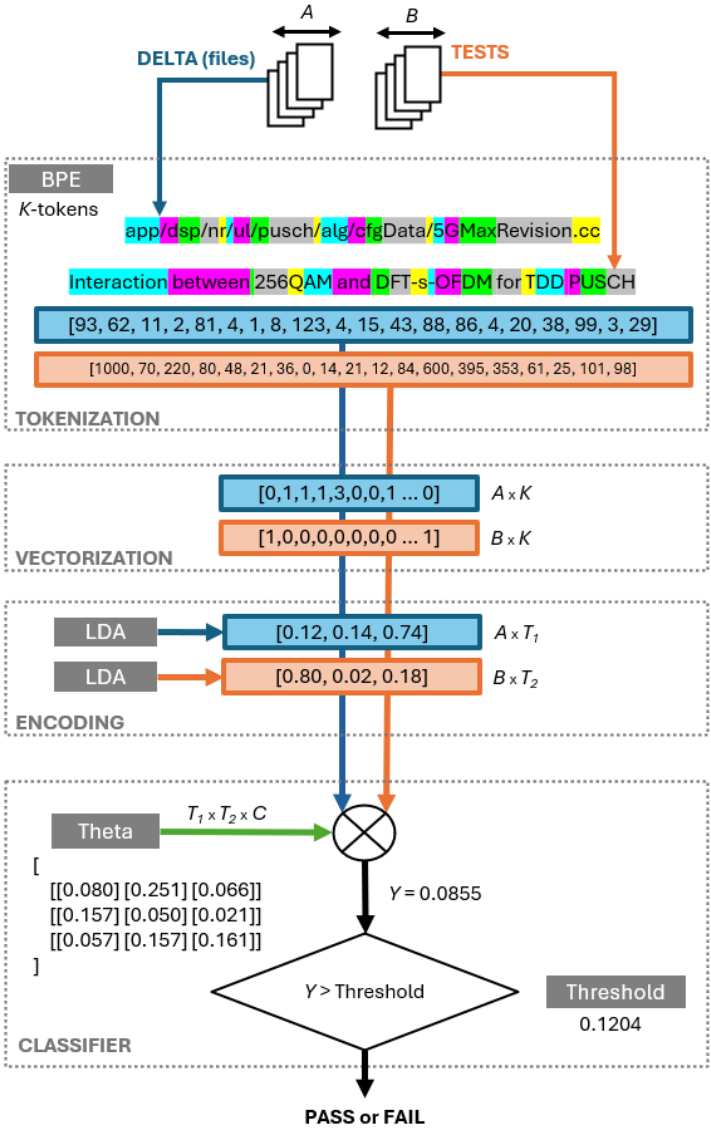


Figure 4.6: Data handling and categorization involve characterizing A-files and B-tests with K-tokens, then aligning them with $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ topics. Multiplying with $\boldsymbol{\Theta}$ yields a probability expressed by C-classes

each representing a unique identifier for sequential tokens within the tokenizer's dictionary (e.g., the word "Interaction" is represented with the number 1000).

The algorithm of choice is byte-pair encoding (BPE), first introduced by Gage [42] and later used in GPT models [99]. The algorithm was chosen for its ability to effectively handle OOV words by breaking them down into known sub-words. This is crucial in our domain, where file and test names often contain specific, compound technical terms and acronyms that would not be present in standard vocabularies. The BPE progressively combines the most frequent character pairs in a text to build a desired vocabulary size. For instance, in the text "low lower lowest", BPE may merge "l" and "o" into "lo", then "lo" and "w" into "low" etc. The adjustable parameter is K, a vocabulary size (or number of tokens) such as $K \in \mathbb{N}$. The tokenizer must be trained on two datasets and used exclusively. It is essential to determine if the text is case-sensitive, as abbreviations such as "pusch" and "PUSCH" might be tokenized differently (e.g., p-usch vs. P-US-CH, see Figure 4.6). Variations in capitalization, such as camel case in file names, can result in different tokens, potentially leading to suboptimal outcomes due to inconsistent encoding of identical contexts. For consistency, file paths and test names were converted to lowercase before preprocessing with the BPE tokenizer.

4.2.2 Embedding

Embedding transforms the previously obtained tokens into a numerical format that machine learning algorithms can process. In our approach, we use a method that counts the frequency of each token in the text. Given that each software delta consists of multiple files, we decided to avoid associating tests with files in a one-to-one manner, as this approach would generate multiple (and repetitive) sparse vectors. Alternatively, our method generalizes over the whole set of deltas by breaking down all modified files into tokens. In the given example, the delta (blue vector) includes a token "ul", numbered 1 and occurring once, and a "\" token, numbered 4 and appearing three times, thus assigned values of 1 and 3 in the output vector. Following this, both the test and delta are expressed as distinct N-dimensional vectors, reflecting frequencies of the tokens. Tokens present in the dictionary but absent in the text are assigned the value of 0. This configuration is

visualized in Figure 4.6 as N-dimensional token count vectors (the blue vector for the delta and the orange one for the test), where many positions can be zero. This sparse vector format is crucial for the LDA algorithm to allocate probabilities to each topic according to token counts in each sample.

4.2.3 Encoding

To discover the underlying textual structure, we employ LDA, a probabilistic generative model to discover abstract topics in a collection of documents. The formal graphical representation of LDA is detailed in the theoretical background chapter (see Figure 2.10, p.65). The optimal number of topics T, a key hyper-parameter of the model, was determined empirically. A systematic grid search was performed, testing a variety of parameter combinations: the number of topics for the tests and deltas was set to $\{2, 4, 8, 12\}$, and the size of the tokenizer vocabulary was set to $\{100, 200, 500, 1000, 2000\}$. The final configuration was selected based on a Pareto front analysis, optimizing the trade-off between the TPR and TSR metrics. A detailed description of this process and the results for each configuration are presented in Section 4.2.6.

In this practical application, we treat the collection of all unique test names as one corpus and the collection of all unique file paths from deltas as a second separate corpus. We then train two distinct LDA models, one for tests and one for deltas, to learn the specific topics within each domain. The output of each LDA model is a set of probabilities of topics for each document. Therefore, it is a vector of i-probabilities for t-topics, such as $\sum_{i=1}^{T} t_i = 1$, providing a numerical representation that encapsulates the document's semantic content. For example, for a given test, the model produces a vector such as $[0.8, 0.1, 0.1]$, indicating an 80% probability of belonging to Topic 1, 10% to Topic 2, and 10% to Topic 3. These topic probability vectors serve as the semantic feature representations for the subsequent classification step.

The LDA is highly effective in identifying common themes, such as frequent tokens in specific contexts (such as, for instance, signal modulation in the case of 2G to 5G cellular networks). Referring to Figure 4.6, this step encodes input vectors into probability vectors for thematic topic assignment based on token co-occurrence. For instance, the presence of the tokens “Interaction” and “DD”

(from TDD or Time Division Duplex) can result in assigning the vector corresponding to the test name (orange) to the first topic with a probability of 80% (first value in orange vector). Therefore, by grouping tests and deltas thematically, we can link these insights to testing results, assessing whether topic-assigned groups indicate problems, potentially revealing software bugs. Hence, in this study, **we classify thematic contexts and assess the likelihood of bugs when there is alignment between testing and software contexts.**

Being unsupervised, LDA automatically discovers topics based on the statistical properties of the data, particularly token co-occurrence patterns, eliminating the need for manual labeling or predefined categories.

4.2.4 Classifier

After the encoding stage, where both the delta and the test are transformed by LDA into topic probability vectors, these vectors become the *input features* for the final classifier. Figure 4.6 illustrates this crucial step: the output from the "Encoding" block (the topic distribution vectors T_1 and T_2) is fed directly as input to the "Classifier" block. The LMLDA model uses a linear architecture inspired by logistic regression at this stage to compute the joint probability of failure. Given that LDA provides us with probabilistic distributions rather than direct feature values, we construct a joint model that leverages these distributions to predict binary test outcomes (pass/fail or 0/1). As shown in Figure 4.6, we use LDA to create marginal distributions for T_1 and T_2. In this context, T_1 represents the vector for the software delta (changes) and T_2 represents the vector for the test. These vectors are composed of probabilities that indicate the likelihood that each variable (delta or test) is assigned to specific topics. For example, each element in T_1 and T_2 corresponds to the probability that the delta or test belongs to a particular topic identified by the LDA model. Then, the conditional distribution $p = (C|T_1, T_2)$. indicates the probability of C (whether the combination of the test and the delta results in a software error) given the two factors T_1 and T_2.

By considering a delta as a representative for all changes in software, against each test, we form delta–test pairs, where each pair consists of a test and the set of code changes (delta) it is being run against. Hence, a subset of these

pairs can be denoted as w, the number of samples, or batch size. Then, let us define y as the dependent variable, indicating the probability of failure. Let $\boldsymbol{d} \in \mathbb{R}^a$ and $\boldsymbol{t} \in \mathbb{R}^b$ represent the independent variables associated with delta and test, characterized as vectors comprising a and b components, respectively. The classifier parameters are contained within the matrix $\boldsymbol{\Theta} \in \mathbb{R}^{a \times b}$. The final dimension of $\boldsymbol{\Theta}$ corresponds to the number of classes; therefore, it is assumed that the matrix has dimensions (a, b, k), where $k = 1$. Since we are dealing with binary classification, this matrix will be reduced along the final dimension by summing, resulting in y. Consequently, the model is articulated as a sum of products, where $\boldsymbol{\Theta'_{ij}}$ is the result of applying a softmax function to all elements of the original parameter matrix, as defined by Equation (4.2).

$$y = \sum_{i}^{a} \sum_{j}^{b} \boldsymbol{d_{wi}} \times \boldsymbol{t_{wj}} \times \boldsymbol{\Theta'_{ij}} \tag{4.2}$$

Since the output y represents a probability from a regression model, we need to establish a **decision boundary** to interpret the results. For example, if 0 represents a passing combination of test and delta and 1 represents a failure to do so, any predicted y above this threshold would be considered a predicted software failure. According to Grathwohl et al. [49], classifiers can be viewed as energy-based models (EBMs). In these models, each possible configuration of variables is assigned an energy value, where a lower energy indicates a higher likelihood of that configuration. Essentially, the model prefers configurations with the least energy. The loss of cross-entropy in training aims to decrease the energy of the correct class and raise the energy of incorrect classes (thus assigning higher probabilities). Now, the output of such model is called logits, which represents an unnormalized log probability (also defining an energy landscape across the input space). The logits are considered negative energy values. If we assume that some variable X (not related to other variables in this article) has a specific value $x \in X$ with a distribution $P(X)$, then an individual x is a scalar of "energy" in the EBMs. In standard probability distributions, the $\sum P(x)$ for all x equals 1. However, in EBMs, these energies are unnormalized probabilities, so their sum does not equate to 1. For the purpose of normalization to valid distribution, many traditional classifiers conclude with the softmax function. In our scenario,

summing these output energies would yield a normalization constant, denoted as Z, enabling the conversion of energies into probabilities. However, calculating this sum is typically resource-intensive. Alternatively, the Boltzmann softmax operator can transform unnormalized energies into probabilities by taking the exponential of the negative energies and normalizing them using a temperature parameter, which adjusts distribution smoothness. This operator results in a **smooth approximation of the decision boundary**, a concept explored in various studies [8] [93], which we also adopt. In practice, for our LMLDA model, applying this concept means that instead of setting a rigid decision threshold (e.g., 0.5), we establish a smooth, probabilistic boundary. This is particularly beneficial in the context of our imbalanced data, where failures are rare events. Such a "soft" boundary allows the model to more sensitively identify potential failures even when the predictive signal is weak, leading to a higher recall for hard-to-find bugs.

In conclusion, using a straightforward example related to Figure 4.6, the sum of all values in $\boldsymbol{\Theta}$ yields 1, indicating that each matrix value represents the weight (or probability) of the intersection of topics T_1 and T_2 resulting in the probability of finding a software bug. If T_1 is 74% in the third topic and T_2 is 80% in the first topic, the highest coefficient for the final result is the parameter $(3, 1)$, equal to 0.057. Then, as shown in Figure 4.6, conditional probability is $P = 0.0855$. The decision threshold $T = 0.1204$ is determined during the training phase by optimizing the model's performance on a validation set, balancing between false positives and false negatives. In this scenario, since $P < T$, the model predicts that the test applied to the delta will pass (no software bug expected).

4.2.5 Confidence Intervals

To assess the reliability of model parameter estimates, we employ standard statistical techniques for uncertainty quantification. Given the use of maximum likelihood estimation (MLE) and cross-entropy loss, the Hessian and Fisher information matrices are utilized. The Hessian matrix $\boldsymbol{H(\Theta)}$, a matrix of second-order partial derivatives, provides information on the curvature of the likelihood function. The Fisher information matrix $\boldsymbol{I(\Theta)}$, representing the expected value of the Hessian, offers an estimate of the parameter covariance [78]. By inverting the

Fisher information matrix, we obtain the covariance matrix whose diagonal elements correspond to the variances of the parameter estimates. These variances, or standard errors, quantify the uncertainty associated with each parameter. Under straightforward regularity conditions, it is equivalent to the negative expected value of the second derivative of the log-likelihood. Crucially, the invariance property of MLE allows one to calculate the observed information matrix prior to inversion, streamlining the calculation process. After training of the model, the result can be used to determine which parameters of $\boldsymbol{\Theta}$ are statistically significant in predicting the results, thus assessing whether a prediction, with them as the primary coefficient, can be ignored by the process operator (e.g., tester).

4.2.6 Empirical findings

We compare LMLDA with baseline logistic regression, advanced ensemble methods (Random Forest, XGBoost), and decision trees to evaluate predictive precision, efficiency, and scalability. The unsupervised clustering approach, described in Section 4.1, served as an important preliminary study and a baseline that confirmed the potential of delta-based analysis. However, due to its documented underperformance compared to supervised models [141], it was not included in this final comparative evaluation.

The main evaluation of the LMLDA model was performed against other popular supervised classifiers. To ensure a fair comparison, all models (including LMLDA, Random Forest, XGBoost, etc.) used an identical set of input features: the topic probability vectors generated by LDA. This ensures that any performance differences arise from the model architectures themselves, rather than from the data representation.

During experiments, the examination was conducted for different combinations of topic counts in software tests and changes (deltas). For this, variations of datasets were created with 2, 4, 8, and 12 topics, then with 100, 200, 500, 1000 and 2000 tokens, resulting in 80 combinations per classifier. Training was performed on a single NVIDIA GeForce RTX 2070 (8GB) for 1000 epochs with a batch size of 256 and a learning rate of 1e-4, taking approximately one hour per classifier with minor variations based on classifier size. The datasets comprised approximately 12,000 unique tests and 100 testing candidates (with over

200,000 deltas) from regression tests performed between January 2023 and the end of 2024, giving approximately 1,200,000 data points. The data set was divided into training and testing sets using a rolling window approach to simulate real-world software development conditions. At the last evaluation data point, the true labels consisted of 2.16% failed cases and 97.84% passed, highlighting the highly imbalanced nature of the data. Starting in January 2024, we used two-week sprint periods as testing sets, with all preceding data serving as the training set. This window was then rolled forward sprint by sprint throughout 2024, with each new sprint's data being incorporated into the training set for subsequent predictions. This methodology closely mirrors the natural evolution of software development projects, where historical data continuously grow and inform future testing decisions.

Statistical validation

First, we evaluated the statistical performance of the LMLDA model, in order to determine whether it provides deterministic results that significantly deviate from random draws. To achieve this, we set a rigorous **p-value threshold of <0.01**. We utilized two statistical methods: the Mann-Whitney U test [83] and Fisher's exact test [40]. The Mann-Whitney U evaluates whether two independent sample distributions differ, without assuming normality, by ranking all observations combined and checking if one group's ranks are higher or lower than the other's. Conversely, Fisher's exact test calculates the exact probability of observed results in contingency tables under the null hypothesis, making it ideal for small or imbalanced datasets, crucial for assessing binary classification in machine learning. We tested our hypothesis on 5,096 random samples. The mean p-values were 0.171 and 9.58e-106, respectively. The Mann-Whitney U test showed variability with 1,695 nonsignificant and 3,401 significant results. In contrast, Fisher's exact test consistently yielded extremely low p-values, indicating a significant deviation from randomness in LMLDA predictions. The results indicate that LMLDA can detect deterministic patterns in domain-specific imbalanced data. Fisher's exact test shows a consistent deviation from randomness, highlighting the reliability of the model, while the Mann-Whitney U test shows some variability in performance. Despite this, the low p-values from Fisher's exact test strongly support

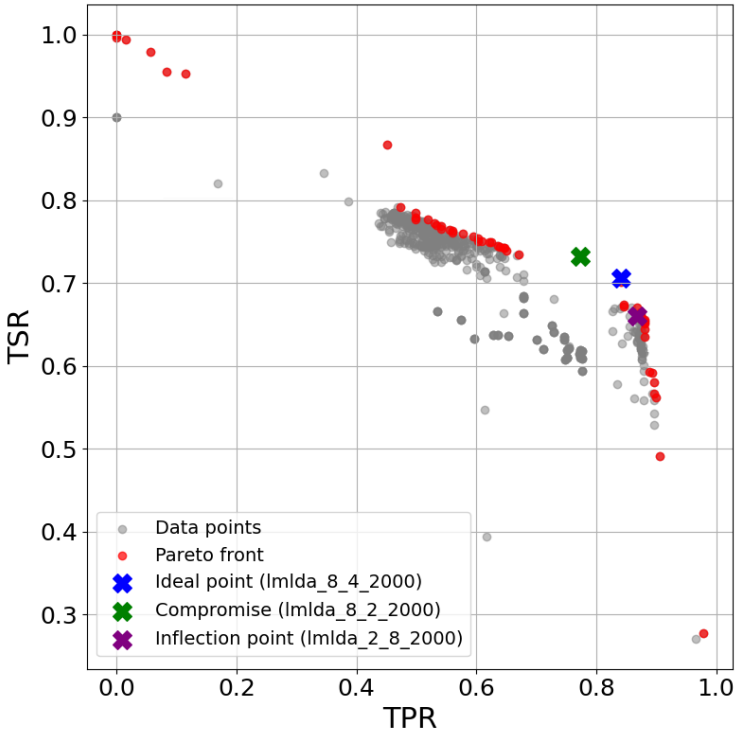


Figure 4.7: Effects of Pareto optimization showing trade-offs between multiple objectives and the resulting front. The model configuration in the legend follows the format 'classifier_X_Y_Z', where X denotes the number of LDA topics for tests, Y represents the number of LDA topics for files, and Z indicates the number of tokens used for text encoding

reproducible results, proving the credibility of the model. The consistent success of Fisher's exact test across multiple comparisons further confirms the model's robustness in complex data scenarios. **LMLDA is capable of learning significant non-random patterns in imbalanced datasets**.

Results

Table 4.2 provides an extensive summary of the classification results. To evaluate the balance between *TPR* and *TSR*, we used a Pareto optimization to

find solutions that represent optimal trade-offs between these competing objectives. A solution is considered Pareto-optimal if no other solution exists that improves one metric without degrading the other. The **theoretical optimal point** ($TPR = 1.0$, $TSR = 1.0$) represents perfect performance, complete test reduction while maintaining perfect failure detection, although this is typically unachievable in practice. We use the Euclidean distance from this optimal point to evaluate Pareto solutions, where shorter distances indicate better overall performance. The inflection point on the Pareto frontier marks where marginal improvements in one metric begin to require increasingly larger sacrifices in the other, while the compromise point represents the solution with minimal Euclidean distance to the optimal point, offering a balanced trade-off. As can be seen in Figure 4.7, gray points represent all the solutions evaluated in the bi-objective space, while the red points highlight the Pareto-optimal front, solutions where no improvement in one objective (TPR) can be achieved without degrading the other (TSR). The Pareto front, denoted in red, generates a set of best solutions from which testing teams can select according to their specific context, choosing higher TPR when test reliability is crucial or higher TSR when resource constraints are essential. This approach enables organizations to make informed decisions based on their risk tolerance and resource availability, providing a flexible framework to optimize test selection strategies.

Analysis and discussion

The LMLDA and logistic regression assume a link between the input features and the outcome log-odds. It works well when the data's true relationship is roughly linear or logistic. When datasets are small or the function $f(x) = \Pr(Y = 1 \mid X = x)$ is smooth and logistic, as in our case, LMLDA and logistic regression outperform other methods, as it estimates fewer coefficients. This is visible on the Pareto front, where the blue, green, and purple X-markers indicate specific configurations that offer distinct trade-offs between the two competing objectives. The optimal configurations typically emerged from scenarios with 8 or 12 topics for the tests, which aligns with the comprehensive results shown in the table. The LMLDA model with 8 test topics and 4 file topics (option containing 2000 dictionary tokens) reached a Pareto-optimal solution, demonstrating $APFD$ of

Table 4.2: Model performance evaluation: cross-topic analysis with averaged performance metrics across token sets (100 to 2000)

MODEL	Topics (tests)	Topics (files)	APFD	ROC AUC	TSR	TPR	FPR	TNR	FNR
Decision tree	2	2	0.655	0.719	0.830	0.601	0.164	0.836	0.399
		4, 8, 12			...				
	4	2	0.660	0.726	0.820	0.626	0.174	0.826	0.374
		4, 8, 12			...				
	8	2	0.640	0.709	0.819	0.593	0.176	0.824	0.407
		4, 8, 12			...				
	12	2	0.675	0.739	0.818	0.653	0.175	0.825	0.347
		4, 8, 12			...				
Random forest	2	2, 4	0.650	0.719	0.843	0.589	0.151	0.849	0.411
		8, 12	0.648	0.717	0.845	0.584	0.149	0.851	0.416
	4	2, 4, 8	0.660	0.726	0.820	0.626	0.174	0.826	0.374
		12	0.655	0.722	0.846	0.592	0.148	0.852	0.408
	8	2	0.645	0.716	0.845	0.581	0.149	0.851	0.419
		4	0.640	0.712	0.845	0.572	0.149	0.851	0.428
		8	0.647	0.716	0.848	0.578	0.146	0.854	0.422
		12	0.650	0.719	0.844	0.587	0.150	0.850	0.413
	12	2	0.656	0.723	0.845	0.596	0.149	0.851	0.404
		4	0.652	0.719	0.845	0.587	0.149	0.851	0.413
		8	0.658	0.724	0.842	0.601	0.152	0.848	0.399
		12	0.657	0.723	0.843	0.597	0.151	0.849	0.403
XGBoost	2	2	0.685	0.740	0.827	0.646	0.166	0.834	0.354
		4, 8, 12			...				
	4	2	0.688	0.740	0.833	0.640	0.160	0.840	0.360
		4, 8, 12			...				
	8	2	0.695	0.746	0.837	0.648	0.156	0.844	0.352
		4, 8, 12			...				
	12	2	0.700	0.749	0.828	0.662	0.165	0.835	0.338
		4, 8, 12			...				
Logistic regression	2	2	0.665	0.728	0.722	0.727	0.271	0.729	0.273
		4, 8, 12			...				
	4	2	0.675	0.739	0.709	0.762	0.285	0.715	0.238
		4, 8, 12			...				
	8	2	0.680	0.742	0.721	0.756	0.272	0.728	0.244
		4, 8, 12			...				
	12	2	0.690	0.750	0.714	0.780	0.279	0.721	0.220
		4, 8, 12			...				
LMLDA	2	2	0.600	0.615	0.760	0.466	0.237	0.763	0.534
		4	0.615	0.624	0.812	0.433	0.185	0.815	0.567
		8	0.680	0.703	0.625	0.776	0.369	0.631	0.224
		12	0.690	0.711	0.576	0.841	0.419	0.581	0.159
	4	2	0.630	0.656	0.696	0.611	0.300	0.700	0.389
		4	0.720	0.742	0.675	0.802	0.318	0.682	0.198
		8	0.740	0.756	0.640	0.865	0.352	0.648	0.135
		12	0.745	0.756	0.640	0.865	0.353	0.647	0.135
	8	**2**	**0.750**	**0.760**	**0.652**	**0.862**	**0.341**	**0.659**	**0.138**
		4	**0.760**	**0.761**	**0.645**	**0.870**	**0.348**	**0.652**	**0.130**
		8	0.755	0.757	0.635	0.871	0.358	0.642	0.129
		12	0.730	0.738	0.580	0.890	0.413	0.587	0.110
	12	2	0.770	0.762	0.641	0.875	0.352	0.648	0.125
		4	0.765	0.760	0.638	0.875	0.355	0.645	0.125
		8	**0.758**	**0.758**	**0.631**	**0.878**	**0.362**	**0.638**	**0.122**
		12	0.720	0.733	0.569	0.891	0.425	0.575	0.109

NOTE: rows highlighted in green, blue, and purple correspond to points of interest – **compromise**, **ideal** and **inflection** points – on the Pareto front shown in Figure 4.7. Three dots are used to denote minimal or no variation in all values in the preceding row.

0.760 and *TPR* of 0.870 and a *TSR* rate of 0.645. The inflection point ($APFD = 0.758$, $TPR = 0.878$, *TSR* rate = 0.631) indicates where marginal improvements in *TPR* begin to require more and more sacrifices in *TSR*. The compromise point ($APFD = 0.750$, $TPR = 0.862$, *TSR* rate = 0.652) represents the solution with minimal Euclidean distance to the optimal point, offering a balanced trade-off between the two metrics. LMLDA improves as more dimensions are used, capturing complex interactions between features. However, the benefit of high *TPR* is accompanied by an increased false positive rate (*FPR*), which affects *TSR*. A key issue in optimization is illustrated by this relationship: improving detection accuracy often requires examining uncertain data regions, which can lead to confusion between true positives and closely related false positives.

Then, the logistic regression showed progressive improvement with increasing test topics, achieving its best performance with 12 test topics (ROC AUC = 0.750, $TPR = 0.780$), although with lower rates of *TSR* (0.709-0.722) compared to tree-based models. The decision tree model showed moderate performance across different topic configurations, with ROC AUC values ranging from 0.709 to 0.739, achieving its best performance with 12 test topics (ROC AUC = 0.739, $TPR = 0.653$). Decision trees divide the feature space into areas with similar outputs, offering flexibility in capturing complex nonlinear relationships. However, they require more data to effectively learn their structure, which can be challenging in software testing datasets that often lean toward successful outcomes. The Random Forest model demonstrated consistent performance across various configurations, maintaining ROC AUC values around 0.716-0.724, with notably high *TSR* rates (0.820-0.848) but relatively lower *TPR* values (0.572-0.626) compared to other models. Random forests usually perform better than a single decision tree, but they may still fall short of logistic methods in scenarios where the data closely follow a logistic relationship. This is evident as Random Forest achieved only about 0.600 of *TPR* for all configurations. XGBoost exhibited stronger overall performance, with ROC AUC values consistently higher than 0.740, reaching its peak at 0.749 with 12 test topics, while maintaining a good balance between *TSR* rates (0.827-0.837) and *TPR* (0.640-0.662). Tree-based methods such as XGBoost, despite their high *TSR* rates, have a significantly lower *TPR*, making them less suitable for accurate test case selection. Finally, the LMLDA model

proves its precision by showing the highest overall $APFD$ among all other baseline approaches.

Understanding how test cases relate to software changes is a major challenge in software testing. We tackle this by analyzing the decomposition of the model's $\boldsymbol{\Theta}$ parameters, which capture learned probabilistic ties in our setup. Splitting these into probability matrices reveals how the model differentiates between relationship classes, offering an interpretable view of its decision process that surpasses the usual opacity of the neural networks. Our study uses the Pareto-optimal configuration with 8 and 4 topics per tests and deltas respectively, focusing on two matrices: P_0, which represents the learned weights for predicting a 'pass' outcome (the matrix given in Equation (4.3)), and P_1, for predicting a 'fail' outcome (the matrix given in Equation (4.4)). In these matrices, each column corresponds to one of the 8 topics identified from the test name corpus, while each row corresponds to one of the 4 topics from the delta file name corpus. A high value in cell (i, j) of matrix P_1, for example, means that when a test predominantly belonging to test topic i is run against a delta predominantly belonging to delta topic j, the model has learned that this interaction has a high probability of resulting in a failure.

$$P_0 = \begin{bmatrix} 0.0092 & 0.0148 & 0.0258 & 0.0199 \\ 0.0131 & 0.0251 & 0.0433 & 0.0325 \\ 0.0006 & 0.0009 & 0.0016 & 0.0012 \\ 0.0010 & 0.0016 & 0.0029 & 0.0023 \\ \mathbf{0.0528} & \mathbf{0.0840} & \mathbf{0.1466} & \mathbf{0.1137} \\ 0.0008 & 0.0013 & 0.0022 & 0.0017 \\ 0.0474 & \mathbf{0.0786} & \mathbf{0.1390} & \mathbf{0.1037} \\ 0.0034 & 0.0077 & 0.0118 & 0.0094 \end{bmatrix} \tag{4.3}$$

$$P_1 = \begin{bmatrix} 0.0059 & 0.0084 & 0.0155 & 0.0114 \\ 0.0173 & 0.0260 & 0.0492 & 0.0340 \\ 0.0009 & 0.0014 & 0.0026 & 0.0019 \\ 0.0032 & 0.0054 & 0.0096 & 0.0070 \\ 0.0177 & 0.0240 & 0.0479 & 0.0348 \\ 0.0010 & 0.0015 & 0.0028 & 0.0020 \\ 0.0065 & 0.0089 & 0.0178 & 0.0128 \\ \mathbf{0.0799} & \mathbf{0.1320} & \mathbf{0.2359} & \mathbf{0.1749} \end{bmatrix} \tag{4.4}$$

Each matrix reflects the topic distributions of the LDA applied during traning to the test cases and changes. The high probabilities (≥ 0.10) are marked in **red**, while the medium-high probabilities (≥ 0.05) are marked in **orange**. Both matrices demonstrate an interesting left-to-right gradient in probability values, indicating a consistent pattern in topic relationships across different dimensions of the analysis. The lower rows in both matrices demonstrate particular significance, suggesting that certain topic combinations carry more weight in determining the relationship between test cases and software changes. The matrix P_0 shows two main clusters of high probabilities, mainly in rows (and topics related to the tests) 5 and 7, ranging from 0.1037 to 0.1466, highlighting strong topic associations to specific topics of deltas. In contrast, matrix P_1 exhibits a concentration of high probabilities in the bottom row, peaks at 0.2359, surpassing the maximum in P_0. These findings have important implications for understanding the relationship between test cases and software modifications.

To visualize the challenge of classifying high-unbalanced and overlapping data, high-dimensional feature vectors are projected into a two-dimensional space using the UMAP algorithm [85]. Figure 4.8 presents the resulting decision surfaces, allowing a qualitative comparison of how each model partitions the feature space. To interpret these plots, consider the following: the blue and red dots represent individual test-delta pairs from the validation set, corresponding to the “pass” and “fail” results, respectively. The background color illustrates the model’s prediction for any given point in that region, with red indicating a predicted failure and blue a predicted pass. The intensity of the color represents the confidence of the model.

An analysis of the individual subplots reveals different classification strategies:

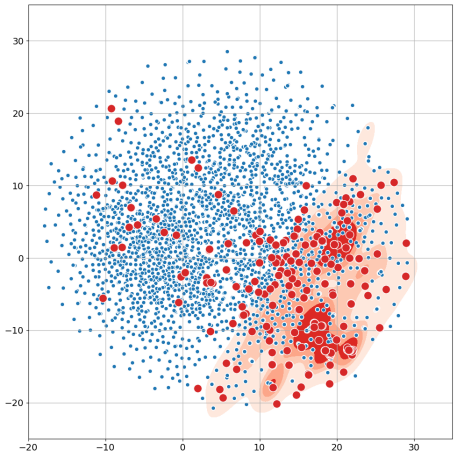
- **Logistic Regression (a)** and **XGBoost (b)** create relatively smooth and conservative decision boundaries. They are effective at identifying the densest clusters of failures but tend to misclassify outliers or sparsely located failure instances (red dots in blue regions). This leads to a higher risk of false negatives for rare but potentially critical bugs. The decision tree and random forest results are not shown, as their behavior is similar to but consistently outperformed by XGBoost.

- **LMLDA (c)** adopts a more aggressive strategy. Its decision surface is more complex and extends further to cover isolated failure cases. Although this correctly identifies most software bugs, it does so at the cost of a higher false-positive rate, as indicated by the larger red prediction areas that cover fewer red dots. This behavior aligns with a strategy that prioritizes minimizing false negatives (missed bugs), which is often preferable in mission-critical testing environments.

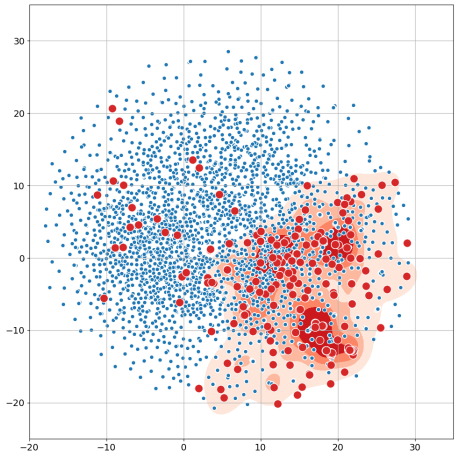
Ultimately, computing confidence intervals for model parameters allowed us to evaluate the statistical significance of each component. The confidence intervals for the model parameters were calculated based on the standard errors derived from the inverse of the Fisher information matrix, a standard technique to quantify uncertainty in MLE models. This analysis, illustrated in Table 4.3, was performed on the Pareto-optimal solution and revealed significant variations in confidence intervals between different parameters.

The parameters listed in the table are the flattened components of the matrix 8×4 $\mathbf{\Theta}$. For example, parameter 0 corresponds to the interaction between test-topic 1 and delta-topic 1, parameter 1 to test-topic 1 and delta-topic 2, and parameter 24 corresponds to the interaction between test-topic 7 and delta-topic 1.

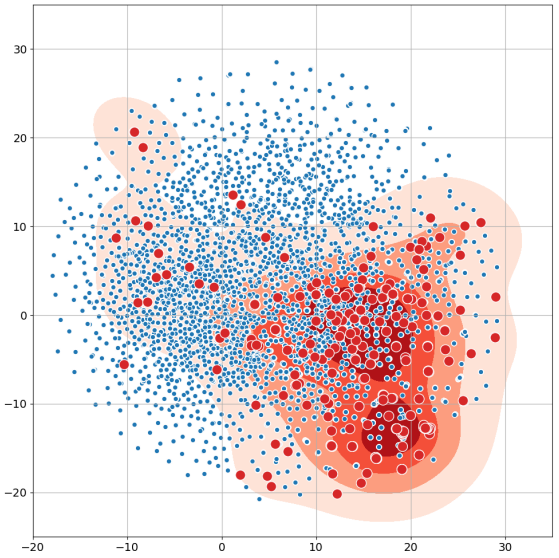
The practical implication of this analysis is two-fold. First, a statistically significant large positive value (for example, for θ_{28}) confirms that the corresponding topic interaction is a strong predictor of failure. In contrast, a large negative value (for example, for θ_{24}) indicates that an interaction is a strong predictor of a "pass" outcome. This allows for identifying which specific the-



(a) Logistic Regression



(b) XGBoost



(c) LMLDA

Figure 4.8: Decision density surfaces for selected classifiers. The red dots indicate software failures. The x- and y-axes represent the two dimensions derived from the UMAP algorithm, a non-linear dimensionality reduction technique used here to visualize the complex structure of the high-dimensional data in a 2D plot.

Table 4.3: The significance of parameters as identified by Pareto-optimal solution in LMLDA model

	Value	**Confidence Interval**	**Proportional Standard Deviation**	**Upper Bound**	**Lower Bound**
0	-2.269	0.004	0.002	-2.265	-2.272
1	-0.993	0.006	0.003	-0.987	-0.999
2	-0.548	0.012	0.006	-0.537	-0.560
3	-1.516	0.005	0.003	-1.511	-1.521
4	-1.572	0.004	0.002	-1.568	-1.575
5	-0.727	0.006	0.003	-0.720	-0.733
6	-1.125	0.009	0.004	-1.116	-1.134
7	-4.048	0.002	0.001	-4.046	-4.050
8	1.905	0.031	0.016	1.936	1.873
9	1.461	0.019	0.010	1.481	1.442
10	1.338	0.016	0.008	1.355	1.322
11	-0.398	0.002	0.001	-0.396	-0.400
12	1.819	0.029	0.015	1.848	1.791
13	1.801	0.029	0.015	1.829	1.772
14	0.252	0.008	0.004	0.260	0.244
15	-0.764	0.005	0.002	-0.760	-0.769
16	-0.630	0.013	0.007	-0.617	-0.643
17	-1.146	0.012	0.006	-1.134	-1.158
18	0.524	0.076	0.039	0.600	0.448
19	-0.839	0.012	0.006	-0.827	-0.851
20	0.209	0.006	0.003	0.215	0.204
21	0.659	0.010	0.005	0.668	0.649
22	0.791	0.012	0.006	0.803	0.780
23	0.560	0.009	0.005	0.569	0.551
24	-5.829	0.001	0.001	-5.828	-5.831
25	-3.417	0.004	0.002	-3.413	-3.420
26	-1.069	0.023	0.012	-1.046	-1.092
27	-1.566	0.012	0.006	-1.554	-1.577
28	2.176	0.053	0.027	2.229	2.123
29	0.602	0.017	0.009	0.619	0.585
30	1.894	0.102	0.052	1.996	1.793
31	1.521	0.057	0.029	1.578	1.465

matic intersections are most critical for bug detection. Second, parameters with very low statistical significance (wide confidence intervals) could potentially be pruned from the model to increase computational efficiency without degrading performance.

The high confidence parameters (> 0.005) demonstrated robust statistical significance with narrow confidence intervals (parameters 0, 4, 7, 11, 24, 25). The moderate confidence parameters, located in the range $[0.005; 0.015]$, showed acceptable statistical significance with intermediate intervals. Then, the low-confidence parameters (≥ 0.015): exhibited broader confidence intervals, suggesting potential areas for model refinement (particularly parameters 18, 28, 30, 31). This hierarchical categorization aligns with the probability distributions observed in the matrices P_0 and P_1, where high confidence parameters correspond to significant combinations of topics. For example, the strong negative significance of parameters 19 and 27 correlates with the elevated probabilities in test topics 5 and 7 for pass cases ($P_{5,3} = 0.1466$ and $P_{7,3} = 0.1390$). Low-confidence parameters might indicate topic combinations that contribute to greater uncertainty in the answer, pointing to areas for model enhancement. Beyond adding data to specific combinations, removing statistically insignificant parameters can increase computational efficiency without compromising model performance. This is particularly important in LDA implementations to prevent topic sparsity in smaller datasets by optimizing the parameter count.

4.3 Scientific and industrial impact

This dissertation presents LMLDA, a new method for TSO that enhances delta-centric grouping of failed tests. It employs LDA for unsupervised data encoding and integrates it with Logistic Regression in a linear framework. This content-aware methodology addresses the challenge of expanding test suites by accurately predicting which tests are most likely to detect bugs based on recent code modifications. The approach was extensively validated in NOKIA's industrial environment through a rolling-window evaluation spanning two years of development data. Although the LMLDA approach shows impressive capabilities for test size reduction ($TRS = 64\%$) with high bug detection accuracy ($TPR = 87\%$), its

most practical and business-critical implementation is found in test prioritization. Such results prove problematic in production environments, particularly in the development of telecommunications infrastructure. The inherent risk of missing 20% of potential defects is commercially unacceptable, as these undetected issues could surface in subsequent testing phases or, more critically, in deployed network environments, resulting in significant customer impact and corrective action costs.

Consequently, research should transition towards industrial application by prioritizing tests over reducing test suites, using the LMLDA model. The primary objective is to achieve a **left-shift** in the defect detection curve, maintaining a comprehensive test coverage while accelerating the discovery of critical defects. By using a dual-criteria ordering system that merges LMLDA's failure probability predictions with past test duration data, the organization ensures early bug detection while maintaining extensive coverage. Tests of shorter duration are prioritized and placed at the beginning of the queue, following the principle that these should be executed first. To ensure thorough validation, data was collected and then analyzed from multiple testing lines and development teams over a full calendar year of production testing (2024), covering both regular releases and emergency patches. It should be noted that the assignments of tests to test lines remains the same. This is because hardware scheduling optimization is beyond the scope of this research and requires a more intensive organizational effort, which is expected to affect the other phases of testing. This approach is enough to fit into existing testing procedure and to ensure that all tests are executed while optimizing the sequence to identify potential issues earlier in the testing cycle, providing the organization with extended time for root cause analysis and defect resolution.

To evaluate the practical impact of prioritization of tests in an industrial setting, the rate of detection of defects over time was compared. Figure 4.9 illustrates this by plotting the cumulative percentage of detected failures against the testing execution time, averaged over a full calendar year of production data. To interpret the chart, observe the two curves: the red line represents the original, unprioritized test execution order, while the blue line shows the optimized order of LMLDA. Empirical results from real-world industrial appli-

cation of LMLDA demonstrate **significant improvements in anomaly detection efficiency**. A significant"left-shif" of the blue curve relative to the red curve indicates an acceleration in defect discovery, meaning that bugs are found earlier in the testing cycle. It allowed 40-60% defects to be identified in the initial testing hours.

Because of the simplicity of ordering mechanism it differs from expected 80% of *TPR*, the impact of precisely finding a high-priority defects is particularly noteworthy. Although acceleration of overall detection is valuable, the primary business objective is to find the *most critical* errors sooner. To measure this specific impact, Figure 4.10 focuses exclusively on high-priority defects as classified by the NOKIA standard severity system. The histogram shows the distribution of time saved for these critical issues. Each bar represents the number of critical bugs found a certain number of hours earlier with the LMLDA-prioritized queue compared to the original test run. This provides a direct measure of the model's ability to reduce the resolution time for the most impactful software failures.

Following NOKIA's standard severity classification system, 98% of high-priority defects were detected earlier, with an average time reduction of 8 hours (one working day) and in some critical cases up to 24 hours earlier than traditional approaches. These substantial improvements in detection timing, achieved in actual telecommunication software development rather than in controlled experiments, validate the practical viability of this approach. This demonstrate that prioritization through LMLDA provides a more pragmatic and implementable approach than strict test suite reduction, especially in mission-critical industrial environments where complete testing and rapid defect identification are essential. This provides strong evidence for the thesis's central thesis. It is shown that an **uncertainty-aware** (via statistical significance of parameters) and **proprietary** ML model can be integrated into the CI/CD pipeline to make anomaly detection more efficient. This result validates the hypothesis that such frameworks not only are academically interesting but also provide measurable improvements in a real-world industrial setting.

From a machine learning perspective, the contribution of this work lies in proposing and evaluating a novel, yet straightforward combination of established techniques in a challenging industrial process. The explicit avoidance of the

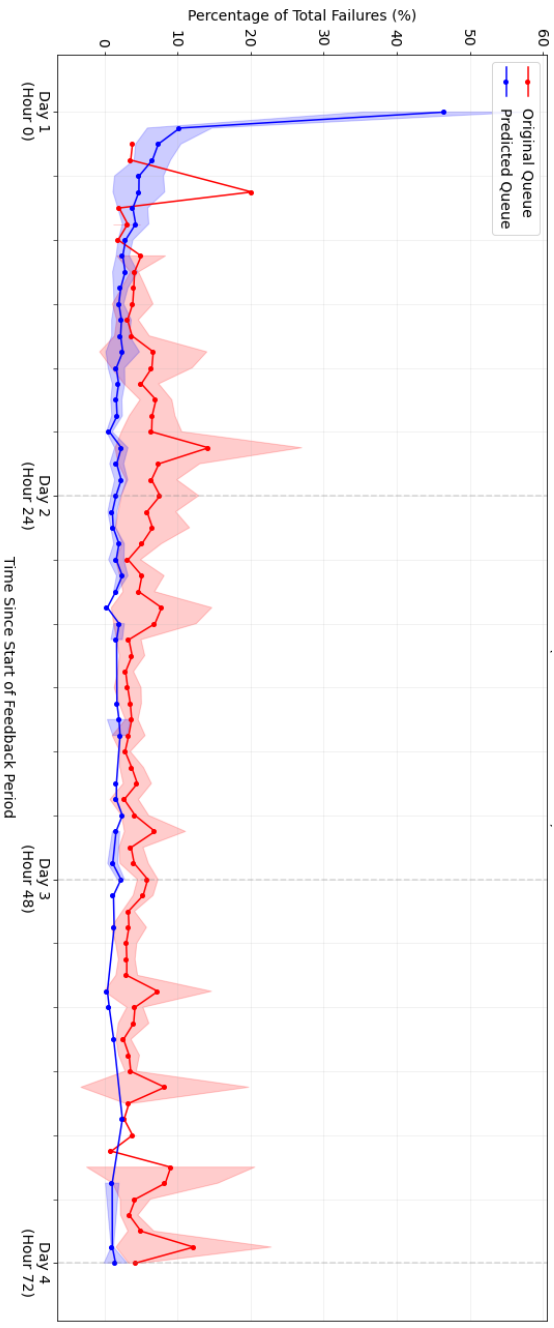


Figure 4.9: Year-averaged hourly test failure distribution (normalized to 100%) comparing original vs LMLDA-optimized sequences, with confidence bands. x-axis shows mean failure rates per execution start hour

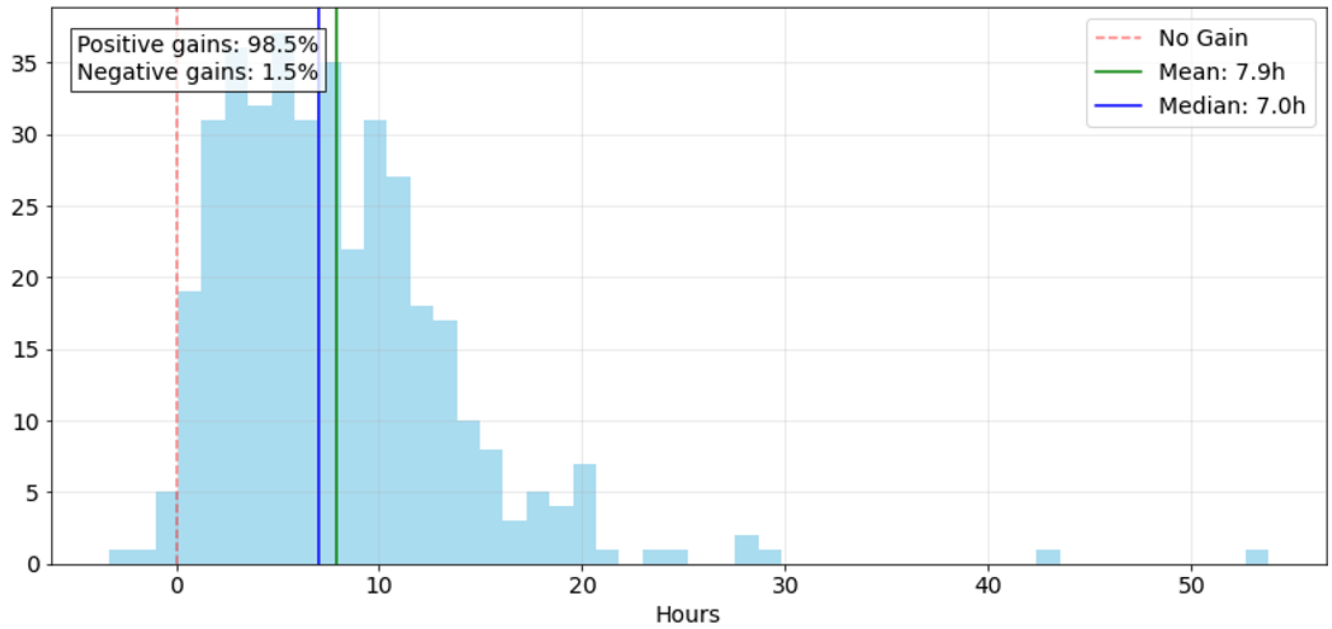


Figure 4.10: Time savings distribution for the most critical errors in the dataset when prioritizing with LMLDA

resource-intensive and often black-box nature of LLM provides a transparent and interpretable approach. It demonstrates how natural language data can be transformed into actionable insights for predicting bug-prone areas. The suggested approach takes advantage of the capabilities inherent in probabilistic modeling. It simultaneously addresses the interaction between testing and code modifications by employing an adapted version of the Logistic Regression model.

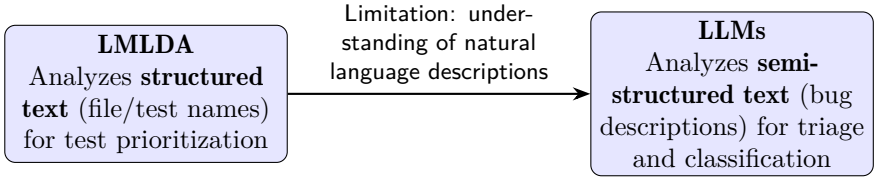
Then, from a software testing perspective, this chapter presents a novel approach to optimize the composition of the test suite. Using NLP techniques, it shows that even basic textual data, such as test names and file names, can be used effectively for TSO. This data-driven method focuses on the correlation between code changes and test outcomes, allowing to prioritize test execution based on predicted failure probabilities. The model significantly reduces testing time while maintaining high bug detection rates, challenging traditional testing practices by shifting from arbitrary or blind test execution to targeted test selection based on code changes.

5 Application of the Large Language Models (LLM) for anomaly triage

Building on the foundations of semantic analysis from the previous chapter, this chapter tackles the core challenge of **reliability** in automating anomaly detection, as stated in the dissertation's thesis. It explores the use of LLM to analyze semi-structured bug reports, a task beyond the scope of the LMLDA model. The central aim is to develop an **uncertainty-aware** ML framework that not only automates the classification and triage of anomalies but does so reliably by quantifying its own epistemic uncertainty. This directly addresses the need for a system that can be safely integrated into a high stakes CI/CD pipeline, automating decisions when confident and delegating to human expertise when not.

The research logically progresses from a model that understands *what was changed* (LMLDA) to a system that understands *how the resulting problem is described*, thus addressing a different and more complex aspect of anomaly detection. This progression is visualized in Figure 5.1.

Figure 5.1: Conceptual progression from TSO to use of the LLM



However, the inherent complexity of telecommunications data creates distinc-

tive linguistic patterns in which error descriptions frequently overlap and share common technical terminology between distinct functional domains. This phenomenon is clearly demonstrated in Figure 5.2, which presents a UMAP visualization of the semantic meaning of error descriptions in several functional areas of a cellular base station. In this plot, each point represents a single bug report, and its proximity to other points indicates how similar their textual descriptions are in meaning. The visualization reveals that bug reports do not form distinct, isolated clusters for each functional area. Instead, there is substantial overlap, particularly for security-related issues. For example, a security vulnerability might be described with similar technical jargon whether its root cause is the radio head components or the main motherboard, making it challenging for automated systems to distinguish between them based only on the description. This semantic ambiguity highlights the need for sophisticated models that can grasp deeper contextual nuances.

This chapter presents a systematic examination of LLM applications in textual data related to telecommunications, analyzing the transition from basic contextual comprehension to sophisticated uncertainty quantification. This resulted in the formulation of an advanced pretokenization framework and a context-specific lexicon derived from 3GPP documentation. The research presented in this chapter directly supports the central thesis of the dissertation. By developing methods to improve contextual understanding and, crucially, to quantify model uncertainty, the aim is to build a more **reliable** framework for anomaly detection. This system is designed to be **efficient** by automating the complex task of bug triage and classification, fitting directly into the CI/CD pipeline as a tool that can be trusted in a high-stakes industrial environment.

Section 5.2 starts the initial investigation with the implementation of Bidirectional Encoder Representations from Transformers (BERT) within a Siamese network architecture for label similarity matching. The aim is to identify identical or similar software anomaly descriptions in geographically distributed development teams. This approach is critical for reducing false positives in the dataset and saving significant engineering effort by preventing multiple teams from trying to determine the root cause of the same issue in parallel, which is a common problem in worldwide companies with different time zones and teams.

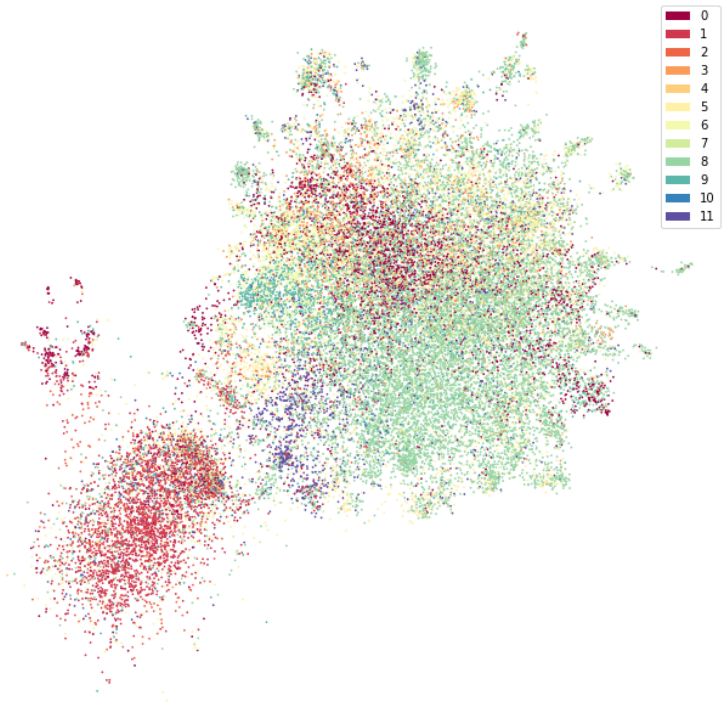


Figure 5.2: Visualization of the encoding of bug descriptions in all functional areas. Each integer value is related to specific functional area of a cellular base station produced by the NOKIA. Tickets form a large and rather uniform cluster, showing how similar they are. Algorithm used – UMAP [85]

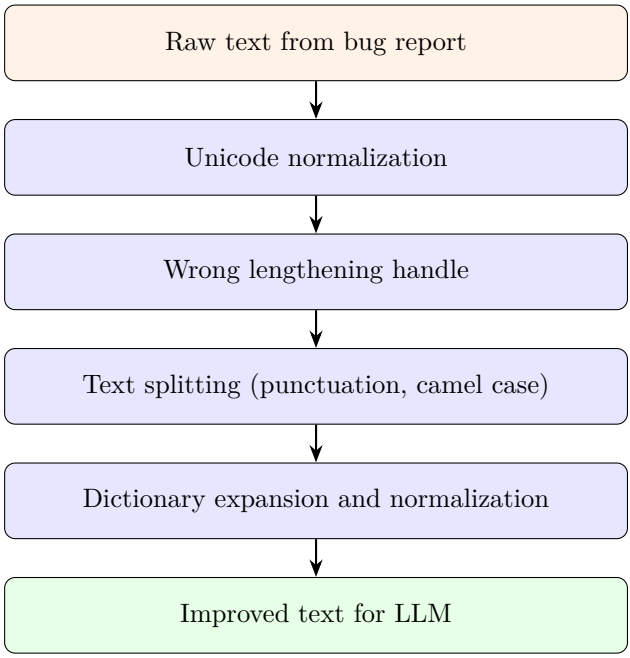
The progression of the research led to the experimentation with a more sophisticated LLM architecture focusing on automated categorization of software anomalies and assignment to development teams. The work in Section 5.3 focuses on fine-tuning protocols (by prompt engineering and the classical approach) to improve the model's understanding of domain-specific terminology. Empirical analysis revealed that technical nomenclature, domain-specific abbreviations, and 3GPP standardization terminology presented significant challenges for conventional LLM implementations. The research led to the design of an automated anomaly detection system that processes incoming bug reports, assigns them to the responsible development team, and improves the CI/CD pipeline. Through targeted fine-tuning procedures, this system demonstrated quantifiable improvements in model performance, directly correlated with improved classification metrics.

The concluding Section 5.4 addresses the fundamental challenge of quantifying epistemic uncertainty and contextual understanding of software anomalies (descriptions and root causes). Building upon the empirical findings from various LLM implementations, this research provides a critical analysis of how different architectural families and their unique attention mechanisms directly influence a model's ability to provide confident predictions with high precision, which is crucial for real-world industrial usability. This methodological advancement has shown particular effectiveness in directly measuring what the LLM model does not know, making the system capable of self-reflection and signaling its uncertainty to the engineer.

5.1 Preprocessing methods for technical corpora

Before any LLM can be applied effectively, the raw text of the bug reports must be carefully cleaned and structured. This preprocessing is critical because the telecommunication corpora are filled with proprietary terms, acronyms, and formatting artifacts that are not found in the general-domain text on which most LLMs are trained. The primary challenge is the transformation of these highly exclusive meanings (acronyms, abbreviations, internal naming) into forms the LLM can use. Figure 5.3 outlines the pipeline proposed for this transformation.

Figure 5.3: The proposed preprocessing pipeline for technical corpora



The proposed solution must be considered as a feedback loop between the dataset and the human operator (ref. connections 'UPDATE' and 'VERIFY' in Figure 2.6). The period after which dictionary maintenance will take place must be specified. The operator must be able to determine the words that should be removed from the dictionary or added as new words. This applies both to obsolete vocabulary (products that are no longer developed by the company) and words whose frequency of occurrence is very low. The dictionary of such keywords must be constantly updated with new words, abbreviations and acronyms. It is proposed that such a period should occur following each significant code release.

5.1.1 Technical corpus

A single word, not serving as an abbreviation or acronym and therefore not extendable, is proposed as the smallest independent component of the constructed

corpus. This word should be associated with several features. The proposition is presented in Table 5.1. Two special tags are introduced, which should later be replaced during the process: `SELF` denotes the use of the same string, while `UNK` indicates that the value is currently unknown. Once this step is complete, it is suggested that all identified words (and those absent from the English vocabulary) are named by their corpus meaning is suggested (i.e., `POC`, a part of the corpus. This becomes handy when searching for general meanings from a corporate perspective (as an analogy for the `POS` tag in the English corpus). For example, it can be used to mark the radio head (as `RFUNIT`) or the company's product names (with a `PRODUCT` tag). Only then can the implementation of an algorithm that uses the new corpus in a data processing step start. Such a method must respond to the needs of cleaning and splitting groups of words into individual entities, taking into account their relationships (e.g., the ancestor being an initial acronym and its children, i.e., all the words that make up its extended form). Ancestors, such as acronyms, can be either retained in their original abbreviated form or substituted with their more extensive meanings, comprising a few words.

Table 5.1: Proposed list of word features

Feature name	**Value**	**Description**	**Example**
raw_form	`SELF`	raw (initial) form of a word	mRRH
normalized	`UNK`	clean version of a word	mrrh
lemma	`UNK`	lemma of a word	mrrh
definition	`UNK`	extended meaning of a word	`UNK`
extension	`UNK`	extended meaning of acronym or abbreviation	micro remote radio heads
origin	`UNK`	indication of word origin	corporate
ancestor	`UNK`	word from which it originated	`SELF`
`POS`	`UNK`	part of speech	`NOUN`
`POC`	`UNK`	part of corpus (only for the corporate dictionary)	`PRODUCT`
stopword	False	should word be treated as a stopword	False

Dataset of such technical origin is expected to contain text originating from

different sources and is characterized by different forms of writing or formatting. Some of the artifacts are caused by just copying text between word processors using different language families, e.g., from Chinese to English. All characters should be normalized according to their Unicode definition. Selecting Unicode is a fundamental decision for data consistency beyond just a technical hyperparameter. In global companies, bug reports and logs often include text from various character sets. Unicode offers a universal standard to manage this diversity and prevent errors that regional encodings might cause. One of the key reasons is that many characters have different notations, for example `\u200b`, `\u202c`, `\uf0d8`, `\x7f`. They are characterized by a common Unicode category `Cc` (control chars) or `Zs` (space separator).

In the course of this research, two additional methods were included to assist in text cleanup. The first is **wrong lengthening handle** – making it possible to repair some misspellings like insu*fff*icient into insu*ff*icient. The second method, used directly after the first, is the **word lengthening filter**. It is used to verify whether a piece of text, presumably a word, is long enough to be taken into consideration.In this particular instance, it was decided to eliminate any segments less than three characters in length. However, some important acronyms or abbreviations were just that short, i.e. UE, which stands for 'user equipment' (a mobile phone). In this special case, which are defined by originating from corporate dataset, such a rule was ignored and word is kept in vocabulary.

These methods ensure that the provided text is clean and ready to be used in further feature extraction. Emphasis should be placed on punctuation marks, as they hold a significant role within this dataset. Not only do they separate sentences or give them emphasis or specific meanings, but they also serve to describe abbreviations or functions in the system logs of the device. To approach this, two additional methods were used. They allow us to find potential words that can be split into more words or left out and excluded from further text processing. The first operation is **splitting text on selected characters**, which are from the Control (`Cc`), Separator (`Z`) and Punctuation (`P`) category in Unicode. The second operation, performed after the first, is **splitting words on the camel case**. It may help in specific cases such as function names from the source code or system logs, but may be tricky when some internal vocabulary consists of

a combination of upper- and lower-case characters (like FHGW which is Nokia AirFrame Fronthaul Gateway or the mRRH, the micro Remote Radio Heads). However, by defining and keeping all such instances in the proposed format (see Table 5.1), a user has full control over deciding whether such words must be extended, left in the original form, or deleted. An example of the transformation of the title of the ticket can be found in Table 5.2. Noticeably, the mRRH word may be easily expanded to introduce additional context to the sentence. It depends on the user's choice.

Table 5.2: Example of a sentence subjected to text preprocessing using the proposed functions

Original form	**Transformed**
CpRtCell	call processing real time cell
configuration	configuration
updating	update
failure	failure
after	– (possible stopword)
mRRH	mrrh / micro remote radio head
reset.	reset

5.1.2 Prior distribution of the classes – labels smoothing

As defined in Chapter 2 and shown in Figure 2.6, the problem of constantly changing sample distribution is related to the tendency of the model to favor the most frequent classes in the dataset. This problem is addressed by using the historical data of each department's participation in previously resolved bugs. Until the bug is fixed, it happens that a ticket is passed between departments. Usually, one department asks another for an additional consultation. An example would be when a development team working on a new FPGA decides to ask a group of programmers associated with the radio heads about interfaces between their software components. Although ultimately, just one (class c label) will be used as the answer, two distinct domains are present. This happens when engineers are not sure about the root cause of bug symptoms. Thus, the final probability of the labels should be split between the associated departments. All

of these changes may be treated as a directed graph with loops and perceived as a change in state from one to another. It may be written as a vector in which consecutive nodes are defined as numerical values of transitions between consecutive states (departments). This approach, presented in Figure 5.4, is easy to describe in the numerical form.

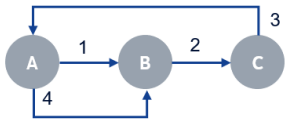


Figure 5.4: An example of a transition graph with a loop – there were different departments required to find out that department B may be the best one to check the root cause of a problem.

Utilizing one-hot encoding for output labels (characterizing a categorical distribution) results in a zero-one vector where the value one (and consequently the entire probability) is allocated to the department (class) where the bug was resolved:

$$[\,\mathbf{0}\quad\mathbf{1}\quad\mathbf{0}\quad 0\quad 0\quad 0\quad 0\quad 0\,]$$

Consequently, other classes with suspected symptoms are overlooked. Thus, utilizing prior knowledge, it can be converted into a more meaningful form:

$$[\,\mathbf{0.20}\quad\mathbf{0.70}\quad\mathbf{0.10}\quad 0\quad 0\quad 0\quad 0\quad 0\,]$$

This is the form of the operation known as the 'label smoothing', originally proposed by [114]. The basic problem of this solution is to find the optimal way to determine these values. A minimum constant threshold is proposed for the department where the bug was ultimately resolved (specifically, the one directly involved in the bug correction process), such as a value of $p = 0.5$, with the remainder being distributed among all nodes in the graph referred to previously. Although such a solution introduces higher uncertainty in the data, it reflects the actual decisions of the engineers in the company.

5.2 Semantic similarity of anomaly reports

One of the primary challenges in this process, which can be named the problem of **duplicate anomalies between sites**, emerged from the global presence of NOKIA. Testers in different offices worldwide might encounter and report the same error, creating duplicate tickets. The current process risks that multiple coordinators, especially those working in different time zones, miss out on these duplicates. This highlighted the need for an automated similarity check during ticket creation, which would not only reduce coordinator workload but also ensure consistent bug fixing across all affected products and versions [142].

The challenge becomes more complex when dealing with software that incorporates third-party tools and libraries. For example, security-related errors in these components require swift action, as they can affect individual products (such as a specific base station version) or entire product lines. A prime example is the OpenSSL library vulnerability (CVE-2022-2274) that potentially allowed remote code execution under specific circumstances. Taking into account the content of the reported problems, some of them are described in a similar manner. It is characterized by an accumulation of problems found in software from other sources than internal:

Ticket A: Info-Zip Unzip 6.10, 6.1c22 -- Multiple Vulnerabilities CVE-2018-1000035, CVE-2018-1000031, CVE-2018-1000032 and more

Ticket B: Info-Zip Zip 3.0, Unzip 6.0 -- Multiple Vulnerabilities -- CVE-2022-0530 and more

Tickets A and B exemplify cases where similar security issues are described with partially matching syntax. The presence of CVE identifiers and similar structural elements makes these tickets relatively straightforward to identify as related. However, the challenge becomes more complex when examining tickets that describe related issues but use different terminology or reference systems, as shown in the following examples:

Ticket C: Mozilla Network Security Services (NSS) 3.68, 3.72 -- Remote Code Execution Vulnerability -- 3.68.1, 3.73

Ticket D: `Python Package: lxml --lt 4.6.5 -- Local Improper Input Validation Vulnerability -- GHSA-55x5-fj6c-h6m8`

The adaptation of domain knowledge, particularly in the telecommunications sector, presented a significant challenge. The bug descriptions in NOKIA's 5G mobile network base stations are heavily filled with technical jargon, abbreviations, and company-specific terminology that rarely appear in general English vocabularies. The situation becomes particularly evident when examining the provided ticket examples. Although some tickets share obvious syntactic similarities, others might be semantically related despite having vastly different textual content. This requires a system capable of understanding context through subword information and contextual relationships.

This challenge presents both business and research opportunities. From a business perspective, early identification of similar tickets reduces duplicate engineering efforts and improves resource allocation. From a research point of view, it offers the opportunity to develop more sophisticated classification systems that reduce false positives and better handle the complex nature of software anomaly descriptions.

Eventually, the following research question must be answered, specifically **RQ 3**: How effectively can a fine-tuned Siamese network architecture identify semantically duplicate bug reports within a specialized telecommunications domain characterized by high lexical overlap?

5.2.1 Methodology

All textual parts of the ticket are merged into a single sentence to represent it as a data sample. The text is then tokenized. In this case, a data vector usually has between three and five hundred tokens. Only 2% of the entire history of software bugs are known as very similar items (duplicates). Such are defined by the description of symptoms that indicate the same bug, if detected in the same product with the same software version, on the same hardware configuration. The resulting subsets usually contain 2 to 5 reports linked to each other. One ticket is defined as a parent, and the rest are attached to it. The problem is visualized in Figure 5.5.

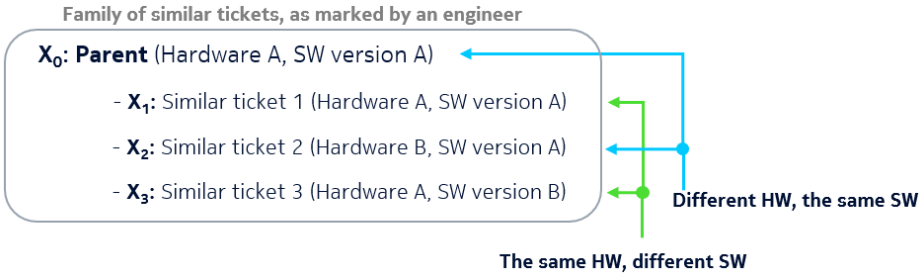


Figure 5.5: The explanation of the different types of similarity between tickets is marked as such by an engineer. The parent ticket X_0 is at the same time similar to X_1 (by having the same categorical values) and to ticket X_2 even if the following was found on different hardware configuration

The relatively small number of similar tickets is a positive fact for the company but not for research problem. Ironically, by proposing this solution, the aim is to reduce the small corporate dataset even further. For this reason, the final dataset must be specially prepared to compensate for a deficiency of training samples. Therefore, there is a need to artificially generate a larger number of pairs, which will allow the neural network to learn the relationships between selected text fragments. To achieve this goal, six types of ticket pairs were defined, which were then extracted from the initial data set and combined into separate data sets. The rules of creating them are as follows:

- Set A – tickets are combined only from the internal subgroup (related to the parent ticket, but without it);
- Set B – as in set A, but along with a parent ticket;
- Set C – tickets are combined between different subgroups (having different parent tickets, without those parent tickets);
- Set D – as in set C, but limited to the same hardware and software release version,
- Set E – tickets are randomly combined between parent tickets and those that did not have similar tickets (these are undoubtedly distinct bugs);

- Set F – tickets are combined between different subgroups (having different parent tickets).

Sets A and B are composed of pairs with a positive similarity relation, that is, pairs of tickets that are similar to each other. Sets C through F are sets of pairs of tickets that are different from each other. The test set is the same for all subsets and contains unique pairs of similar tickets (created analogously to set B). Consequently, it is possible to ascertain which approach produces better results and to decide if the network should specialize in evaluating the similarity or dissimilarity of the provided inputs. There is an important fact to notice: sets E and F are created for validation purposes to check whether such neural networks actually learn anything and are able to distinguish between different tickets grouped in semi-random or manually controlled process.

The logic of this pairing strategy is illustrated in Figure 5.6. The diagram shows two "families" of similar tickets, each rooted in an original parent ticket (for example, X_0**,** Y_0, shown in blue), which is the first report of a specific bug. The subsequent reports identified by engineers as duplicates are child tickets (e.g., X_1**,** X_2**,** Y_1, shown in gray). The arrows illustrate how pairs of tickets are created to form datasets with either a positive (similar) or negative (dissimilar) relationship, which is essential for training the network.

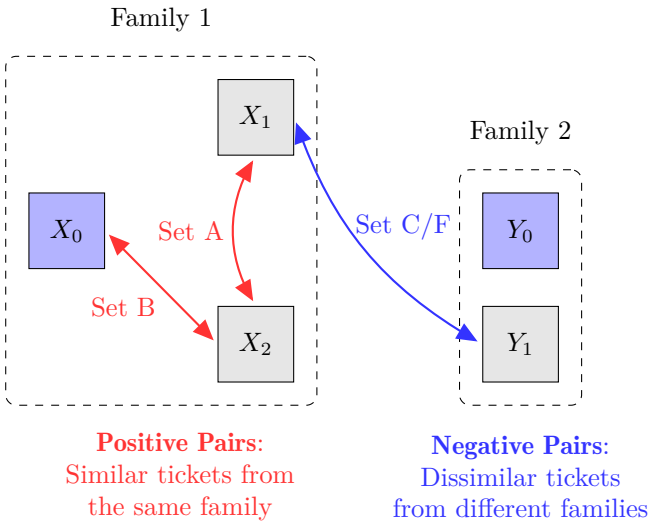
In the result, each version of the dataset was approximately three to four times larger than the original. The next step is to divide them into training, validation, and testing subsets. The naive approach (taking random samples up to a defined amount) could eventually lead to data leak (pairs generated from the same problem could be split between training and testing sets). To avoid such a situation, the following short procedure is used.

1. The dataset must be divided into smaller groups (grouped by parent problem) – groups (or clusters) must be treated as "whole samples".

2. Clusters must be sorted by the date of correction of parents: the latest problems at the top, to put most of them in the testing or validation subset– to evaluate the model against the current company's technological backlog.

3. Then, clusters may be separated into desired subsets.

Figure 5.6: Visualization of the dataset pairing strategy



In relation to the procedure, the initial dataset was divided as follows: 10% of the most recent samples (in terms of when they occurred) were selected for the test set. The pairs in this set were selected as in proposal B, because of their form resembled reality (that is, because they were marked by the testers). In this way, a dataset was constructed that describes the latest trends in the company's software development process and is the same as the decisions of the engineers in the company. Hence, this dataset was not altered or expanded in any way and was used to examine the training results for each dataset from sets A to F. From the remaining portion of the original set, an additional 20% of the most recent pairs were extracted and a validation set was created from them; these samples were selected as in the testing set. From the rest of the samples, successive versions of training sets A through F were created. The overal process of dataset preparation and models training is presented in Figure 5.7

Referring to the research reported in literature and presented in Section 2.5.1 and cost-effective maintenance requirements, BERT encoders were initially selected, then linked in a Siamese network shown in Figure 5.8. Since both en-

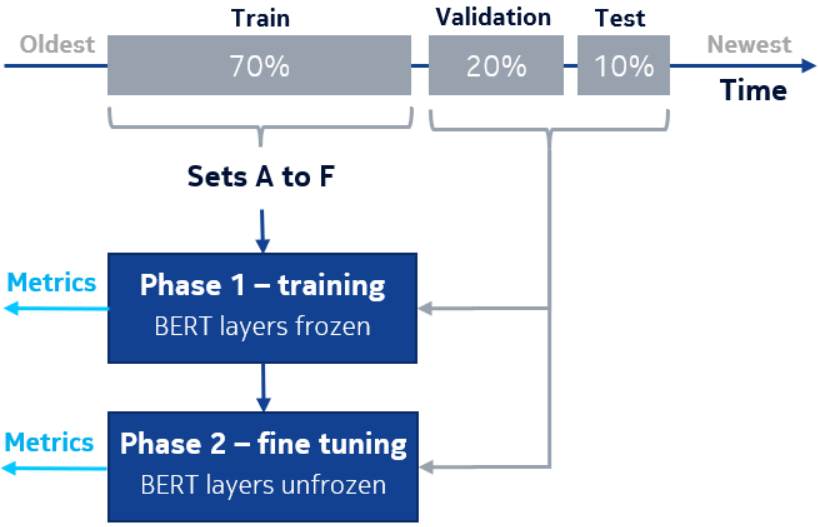


Figure 5.7: The visualization of the process of splitting training dataset into several variations and two phases of neural network training

coders and dense layers share parameters (neurons' weights) with each other, this configuration can be simplified to the use of just one set of them.

For this research, the base uncased version of BERT was used (12 attention heads, 12 hidden layers, each with 768 neurons, that is, a version with 110 M parameters). The model consisted of a single encoding block consisting of a BERT encoder followed by three layers: normalization layer, dropout layer with a rate equal to 0.1, and a dense layer with 64 neurons. The selection of these specific parameters and architecture was based on establishing a robust and methodologically sound baseline. The `bert base-uncased` model was chosen because it is a well-benchmarked foundational architecture, providing a standard measure of performance without the significant computational overhead of larger models. The subsequent layers were designed for function: the dense layer projects high-dimensional output into a more compact 64-dimensional space, which is efficient for calculating cosine similarity, while the dropout layer with a moderate rate of 0.1 serves as a standard regularization technique to mitigate overfitting by preventing the model from relying too heavily on any single feature.

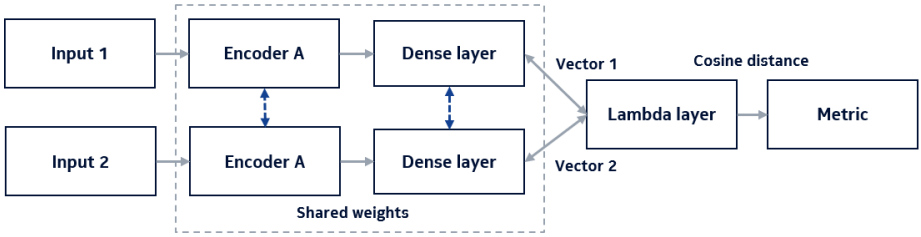
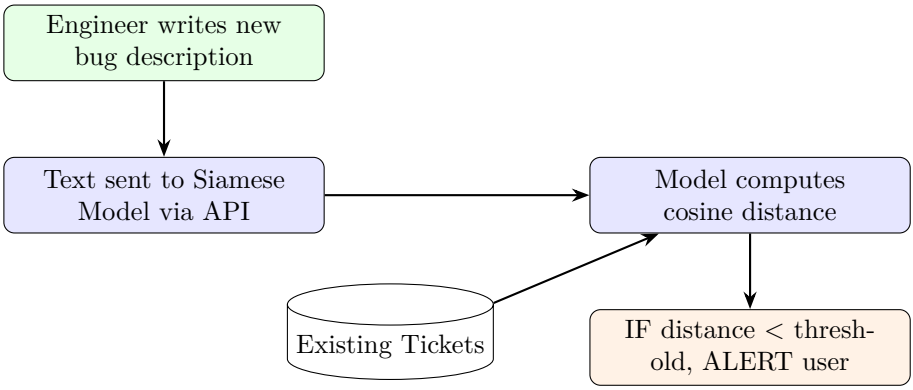


Figure 5.8: Generic Siamese network architecture with shared encoder and dense layer weights

To compare a pair of tickets, two input layers are used. The encoder is terminated by a deep trainable layer, which learns the representation of the vectors after they were pushed through it. The whole network is finished with a Lambda layer whose purpose is to compute the cosine distance between the obtained encodings. In this process, a loss function called the **contrastive loss** was used. Due to dataset versioning and the need to define the best approach to (dis)similarity of certain pairs, the triplet loss is not yet addressed. The goal is to identify more useful text sample relations (positive or negative). Once the construction of triples is clear, the industrial application should implement the triplet loss. The intuition behind the contrastive loss is that similar vectors are as close to 1 as possible, since $\log(1) \equiv 0$ (that is, the loss is Equal to zero). However, since the network is computing cosine distance, not cosine similarity, the final labels are determined in such a way that $d = 0$ means that two vectors are ideally similar and $d = 1$ means that they are completely different.

The practical application of this research is a real-time duplicate detection system integrated directly into the bug reporting workflow, as depicted in Figure 5.9. When an engineer begins to write a new bug report, the system continuously sends the draft text to the trained Siamese model. The model compares this new description against an indexed database of existing unresolved tickets. If a ticket with a cosine distance below a predefined similarity threshold is found, the user is immediately alerted with a link to the potential duplicate. This preventive measure is crucial for saving engineering effort by stopping redundant analysis before it even begins.

Figure 5.9: Proposed industrial workflow for the Siamese network



5.2.2 Results

Tables 5.3 and 5.4 contain the metrics of testing each set for the process with frozen encoder weights and unfrozen (fine-tuning). The first two columns show the percentage of pairs classified as similar when the neural network was very confident (distance $d = [0; 0.25]$) and confident ($d = (0.25; 0.5]$). The third column is the sum of these results, which can be compared to the fourth column, which contains the rest of the prediction results. The distance $d = (0.5; 1]$ means that the network considered these pairs to be rather dissimilar or completely different from each other.

The first phase of training, which assumes that the BERT encoder layers are frozen, proves that the internal vocabulary related to the problem is difficult to use, and the challenge is to capture much of the important context. In Table 5.3, it is noticeable that the confidence of the answers in the similarity of the samples is quite low. At best, 9% strong confidence is obtained and only 48% of the test pairs are marked as somewhat similar. These results are observed for set E, which represented pairs that were dissimilar to each other and that were selected almost randomly. An additional problem arises because the pairs are combined almost independently of the company's fault management process. Due to randomness, there is no control over exactly which pairs would be formed. Thus, a different draw might change this result. Therefore, it is worth looking at the second-best

result, that is, for set F, in which dissimilar pairs were combined by pairing tickets belonging to other groups (having different parent tickets). Interestingly, for the model with a frozen encoder, it was much easier to learn differences between samples set as such by fault coordinators. However, the results for similar pairs (marked in the same way) are much worse. It makes sense as in this approach a dataset consist of carefully selected different tickets, in which most of the words and their context are easier to distinguish.

Table 5.3: A demonstration of the effectiveness of the neural network depending on the approach taken to create the training set – frozen encoder

Metrics	**[0; 0.25)**	**(0.25; 0.5]**	**[0; 0.5]**	**(0.5; 1]**
SET A	6.70%	29.82%	36.52%	63.48%
SET B	5.56%	35.84%	41.40%	58.60%
SET C	3.56%	30.36%	33.92%	66.08%
SET D	4.32%	35.02%	39.34%	60.66%
SET E	8.76%	48.02%	56.78%	43.22%
SET F	6.50%	41.94%	48.44%	51.56%

The great difference from these results can be observed in Table 5.4. It shows how crucial the fine-tuning phase of LLM is. After training, the neural network was able to identify more than 81% of the test samples given as similar, of which only half a percent was probably similar. Only 17% of the samples could not be classified as related or similar.

Table 5.4: A demonstration of the effectiveness of the neural network depending on the approach taken to create the training set–fine tuning.

Metrics	**[0; 0.25)**	**(0.25; 0.5]**	**[0; 0.5]**	**(0.5; 1]**
SET A	81.12%	0.56%	82.68%	17.32%
SET B	80.34%	0.50%	80.84%	19.16%
SET C	79.92%	0.82%	80.74%	19.26%
SET D	76.82%	0.84%	77.66%	22.34%
SET E	75.40%	0.82%	76.22%	23.78%
SET F	79.54%	0.54%	80.08%	19.92%

By enabling the encoder layer to be trainable, making it possible to learn the actual distribution of the tokens in the datasets, It was observed that this type of neural network is capable of finding similarities between ticket pairs. It is visible when comparing the results for sets A and B (similarities) with the rest (differences); it reverses the situation from previous measurements. The network turned out to be powerful enough to catch small differences in the given pairs of texts and correctly find connections between them. This is especially needed for security-related tickets that consist of several external bug identifiers (like the previously mentioned CVE errors) or important keywords.

Despite designing and evaluating a number of datasets, the procedure for generating a test set is still to be improved. It is still naive from a business perspective, as the algorithm does not pay any attention if samples of each product (e.g., radio head types) are represented equally in the training and testing sets. This can be improved. The need for effort is to limit the size of possible false similarities by applying filters on the so-called categorical values. These are related to the tickets and describe different (and additional) features of the product in which the bug is found. Examples of such are software version (branch name), connected hardware peripherals, testline version, code features under testing, and many different. This is supposed to help in situations as described below.

Ticket G: `SW download failed when IPsec for eCPRI management plane is enabled`

There was a situation where several problems related to the same root cause

were described in different tickets. The problem was introduced internally in the code itself as due to configuration misalignment, and on some rare occasions the TCP tunnel would not be set properly. This was clearly a security issue, disabling the gNB station's possibility to upgrade its software to a newer, improved version. However, each ticket with the same root cause could be considered separate (from a business perspective), since those were created for different products and software versions, sometimes having a few months of a time difference between them. A categorical value filter may improve the general accuracy in such cases.

Finally, an important aspect not addressed in this section is the uncertainty of the response. For data with such specific content, it would be useful to be able to estimate the uncertainty in assessing the similarity of two tickets. Therefore, variational inference is the next step. This will allow the solution to correctly measure an epistemic uncertainty. Eventually, it is the main goal from the industrial application perspective, which is a high-stakes critical system in which it must be applied.

5.3 A study on fine-tuning with prompt engineering

The central question of this research is **RQ 4**: Is linguistic adaptation, measured by perplexity reduction, a sufficient condition for achieving high performance on a downstream classification task when fine-tuning an LLM on a niche, technical corpus?

The experimental setup comprises a dataset containing descriptions of technical software bugs: This domain is marked by a rich blend of natural language, source code snippets, and system logs. This content is saturated with abbreviations and acronyms, some standardized by bodies such as 3GPP [1], but many are internal catalog codes for hardware prototypes (e.g., radio heads, FPGAs) without public reference. This linguistic divergence from the general-purpose corpora used to train public LLMs poses a significant challenge. It is important to understand how distinctive domain-specific content affects a model's classification accuracy and to identify ways to mitigate this effect through adaptation.

5.3.1 Methodology

The primary goal is to transform a general-purpose LLM into a domain-specific classifier. The task requires the model to analyze a bug report and generate a label corresponding to one of five predefined categories:

1. specification of base station,
2. motherboard and its interfaces,
3. user interface and base station configuration,
4. network routing and management,
5. radio heads and communication

To ensure deterministic and valid output, the generation process is constrained by the `outlines` library, which forces the model output to match a specific set of labels. This transforms the classification into a controlled generative task. The adaptation strategy began with the principles of prompt engineering, not for few-shot inference but to construct a high-quality dataset specialized for Supervised Fine-Tuning (SFT). Each record in our dataset consists of a prompt (the bug description) and a desired completion (the correct category label), as shown in the example below:

"`product name: 5G base station \n problem summary: [eCPRI] Invalid memory reference (reason 1 `address not mapped to object') detected by kernel at address `callproc_realtime_ue' \n label: radio heads and communication`"

The objective of this full fine-tuning process is to fundamentally re-calibrate the model's internal weights. By training on thousands of these examples, the aim is to shift the model's underlying probability distributions to align with the specialized vocabulary and structure of our bug reports. For computational efficiency, the base model was loaded in a quantized format. This initial, full-parameter update served as our foundational experiment to gauge the model's ability to learn the new domain.

5.3.2 Results

The first measure of success was to evaluate how well the model learned the "language" of bug reports, a concept quantified by perplexity (*PPL*). A high *PPL* indicates that the text is "surprising" to the model, suggesting a mismatch between the model's learned distributions and the data. As expected, the original, non-tuned model found the domain-specific content highly perplexing. For example, a generic description of security-related bugs yielded a low *PPL* of 5.1, while a log-generated error message like "`deploy failed in step monitor-installation...`" produced a staggering *PPL* of 2866. This confirms that much of our dataset originates from a different linguistic distribution than the model's training corpus.

The complete fine-tuning process yielded a dramatic success in this regard. As shown in Table 5.5, Perplexity (PPL) decreased on average by more than 80% in all categories. The density plots in Figures 5.10 and 5.11 visually confirm this shift, showing the mass of the perplexity distribution moving significantly towards zero. This demonstrates that the model successfully adapted to the domain's syntax and vocabulary.

Table 5.5: Comparison of PPL values for original and fine-tuned LLM for thematic groups of software bugs

Group	*Original PPL*	*Fine-tuned PPL*	*Reduction (Δ)*
1	192.69	33.58	82.57%
2	267.93	39.43	85.28%
3	234.70	33.58	86.64%
4	**292.06**	**61.99**	**78.77%**
5	274.53	45.99	83.24%

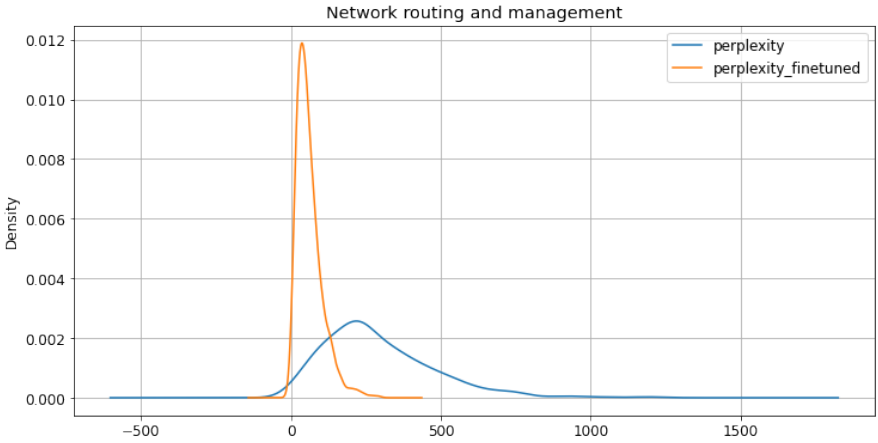


Figure 5.10: Density of perplexity for thematic Group 3, showing a significant shift towards lower values after fine-tuning.

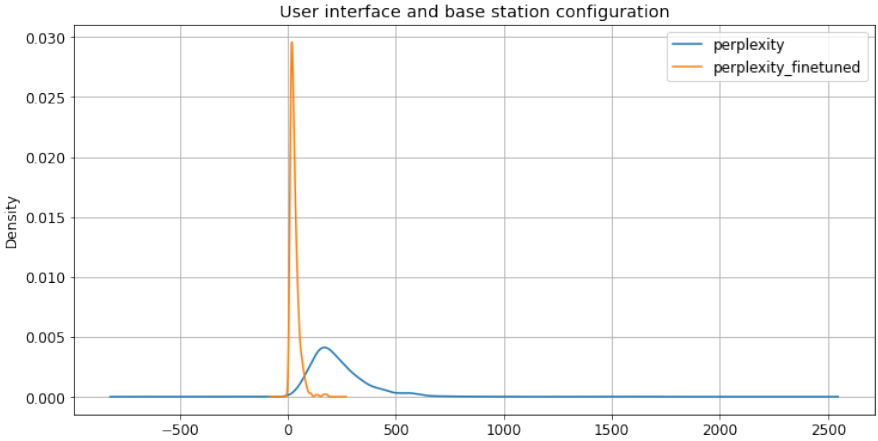


Figure 5.11: Density of perplexity for thematic Group 4, the group with the least, yet still substantial, improvement.

However, this linguistic fluency did not translate into effective classification. The original model showed strong bias, mainly choosing one category (Figure 5.12) and achieving a mere precision 20%. After providing engineered prompt samples, the model predictions became more evenly distributed between

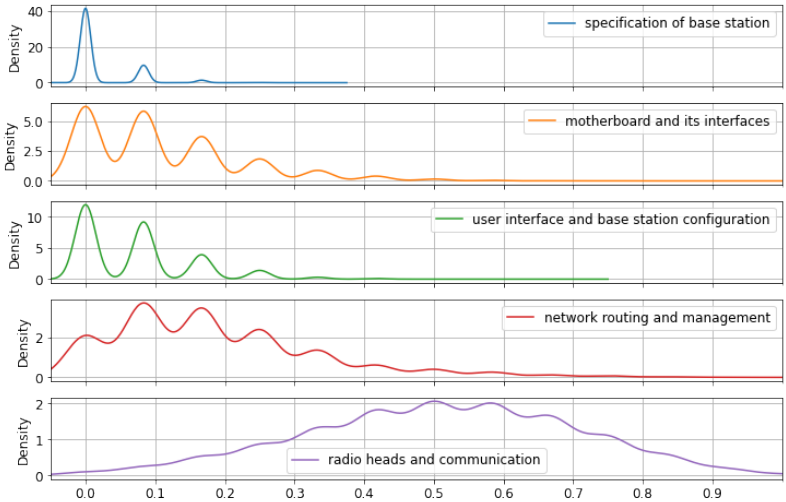


Figure 5.12: Prediction distribution for the original model, showing strong bias towards a single topic.

classes (Figure 5.13), but the overall accuracy saw only a marginal increase to 30%.

The results present a critical insight: reducing PPL is necessary, but not sufficient to achieve high performance in a nuanced task such as classification. While the model learned *what* the words are, it struggled to grasp their deeper contextual and semantic relationships required to distinguish between categories. This gap between linguistic adaptation and task competence points to high **epistemic uncertainty** – the model remains uncertain due to the inherent ambiguity and complexity of the dataset, even after tuning. The primary limitation is that prompt engineering by itself does not modify the model's core parameters, nor does it provide the deeper understanding needed to differentiate between categories. This results in ongoing epistemic uncertainty, with the model's ambiguity and dataset complexity hindering task proficiency, despite clear prompts.

This challenge requires a shift from static foundational models to a more focused fine-tuning approach. A typical but resource-heavy strategy, full fine-tuning, involves updating billions of parameters, requiring heavy computational power and resulting in an static model. In particular, substantial effort only im-

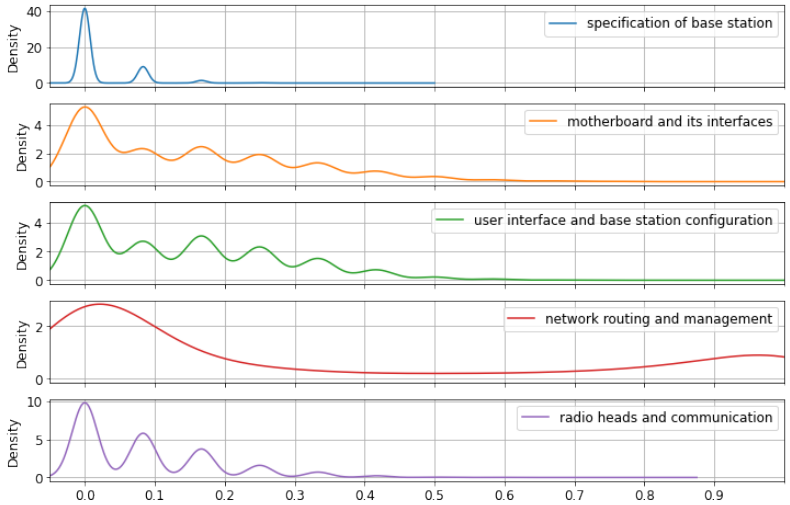


Figure 5.13: Prediction distribution for the fine-tuned model, showing a more balanced but still inaccurate classification.

proves accuracy by a modest 10%, indicating diminishing returns. This prompts the exploration of a more precise and effective method. The PEFT, like LoRA, presents a promising option. It works by keeping the original model weights fixed and adding a small trainable parameter set. This significantly reduces computational demands and targets model behavior tweaks. By modifying the model parameters rather than simply influencing the output, LoRA can surpass the constraints of language adaptation and meet complex classification needs. The subsequent sections will outline its application and success in addressing this specific problem and demonstrate a path to a scalable, maintainable industrial solution.

5.4 A Bayesian framework for epistemic uncertainty quantification

This section introduces a Bayesian framework to tackle this issue by addressing the uncertainty in the classification layer of LLM systems. The analysis is divided

into two parts: initially, Section 5.4.1 examines the foundational models as feature encoders, confirming the approach and identifying issues related to model size and vocabulary. Section 5.4.2 then provides an extensive verification of four modern architectures to develop a reliable, well-calibrated system that is suitable for production.

5.4.1 Preliminary study on LLM as text encoder

Given that the data set contains a mix of both standardized (for example, 3GPP documentation) and proprietary, internal (for example, NOKIA-specific hardware codes) abbreviations and acronyms, it is worth questioning their significance for LLM models. This is interesting, especially from the perspective of the depth of the model and the size of its input. BERT in the medium version works with 768 tokens, while Llama2 works with 4096. The question to be asked is whether the linguistic knowledge possessed by LLM allows it to ignore such rare or new occurrences. Simplifying the dataset by replacing all internal word instances with a uniform word, such as `NOKIA`, presents a new line of exploration. The differences in training a classifier model, using an LLM as an encoder, between the original dataset and its simplified counterpart must be investigated. Then another objective is met: to analyze the differences in training a classifier model, using LLM as an encoder, between the original dataset and its simplified counterpart. This investigation will provide insights into the model's ability to learn from a more abstract data representation.

The first step is to encode the original dataset using both LLMs and to examine the structure of the embedded structure obtained. The embedding matrix obtained in this way was minimized to two dimensions using the UMAP algorithm. Then, each sample is defined by a class assigned to it, each with a unique color associated (ten in total). The result is to be found in Figure 5.14. As can be seen, both models suffer from heavy difficulties in encoding and thus distinguishing the content in the dataset. The samples assigned to different classes intersect, suggesting that the linguistic content within them often exchanges or overlaps.

To address the mentioned research questions, it is necessary to use and carefully observe several metrics. Assuming that the classes related to text samples

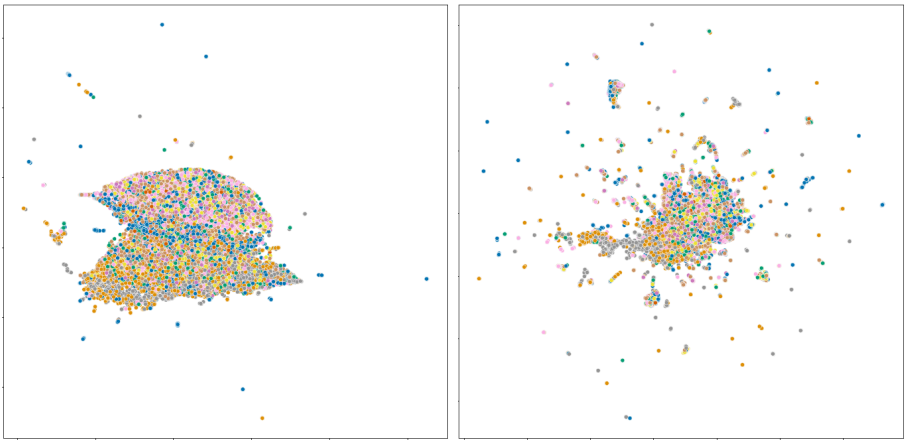


Figure 5.14: Encoding of the original dataset with using a BERT-medium (left) and LLama2-7B (right). There is visible difference in shape of both. It leads to conclusion that single blob with intermingled content is a common result, implying high internal semantic similarity of samples

describe the partitioning of data into groups, the metrics associated with data clustering evaluation can be used. This approach allows evaluating the distribution of samples in terms of general structure (silhouette score and DBI) and PPL.

The initial metrics have been calculated and are presented in Table 5.6. The values obtained confirm what was observed in Figure 5.14. Observably, the Silhouette score is lower than zero in any case, which means that the clusters overlap. For DBI, the best score is zero because the metric is oriented to find clusters that are farther apart and less dispersed. The score of this metric indicates that the groups of texts are rather dispersed from the centers of their clusters, especially for the Llama encoding. It is noticeable that simplification of the dataset leads to an even higher dispersion of the structure. However, from a linguistic perspective, the transformation of a data did not change the outcome that much. A simplified version is related to an increase PPL, indicating the potential value of internal vocabulary, even if a LLM does not recognize it.

The results of this preliminary experiment are shown in Table 5.7. The weights regularization obtained by using the Gaussian distribution (used in modeling

Table 5.6: Preliminary look on LLMs encoding quality metrics

Model	**BERT-medium**		**Llama2-7B**	
Dataset	original	simplified	original	simplified
Silhoulette	-0.087	-0.067	-0.087	-0.079
Davies-Bouldin	30.306	54.231	28.818	81.046
Perplexity	5.127	5.308	4.198	4.302

of classifier deep layer's prior weights distributions) gives confidence that the overfitting problem is avoided. An expected and also confirmed result is the observation that the model with a larger and wider context (Llama2) eventually achieves significantly higher classification metrics. Llama2 outperforms BERT both for the first response score (+10%), the first three responses (+6%) and the F1 metric (as much as +15%).

The accuracy-related metrics are severely limited and are quickly established at a secure, almost constant level. In addition, note the confidence of the predictions. The changes in the mean percentile difference between the probabilities of the classes are visible in Figure 5.15. Llama2 gives more accurate answers, but these are less confident than the answers given by BERT. Notable for Llama2 is the higher average distance between the selected response probability percentiles, implying a bigger difference between the probability of selecting one of the ten classes. **Hence, it is observed that a model with a smaller depth and narrow input becomes more confident than a larger model**.

An important aspect related to neural network training is the convergence to a local optimum. The LLM is used as the encoder, the whole model is only slightly affected by the non-existence of a unique vocabulary. There is no big difference in the training outcome despite the use of an original or simplified dataset. **Eventually, knowledge possessed by LLM leads to identical results, but for the simplified dataset these are achieved slower**. In other words, the unique words are not necessary, but help the LLM-based classifier to find its optimum earlier. As presented in Table 5.7, the network using the BERT model found its optimum much faster than Llama2, without improving its performance in

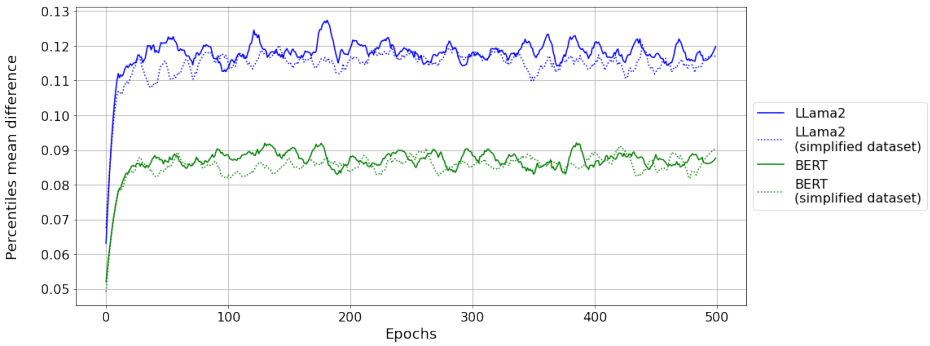


Figure 5.15: Preliminary training phase – changes in mean distances of prediction percentiles between $P_1 = 97.5\%$ and $P_2 = 2.5\%$

the next several hundred iterations. Thus, it makes little difference for BERT whether the dataset contains a larger number of unknown phrases. LLama2 is able to work with the same accuracy for both versions of the dataset, but due to the absence of unknown but highly contextual phrases or words, it needed almost three times as many optimization steps to achieve similar results. This is supported by the **perplexity** metric. The change in a metric during a training can be found in Figure 5.16. The lower perplexity score suggests a deeper understanding of the context, as predictions are more precise. The simplified version of dataset is linked with slightly higher perplexity, suggesting that removal of potential unknown words (not tokens), despite being a total mystery for the model, makes the solution a little bit less confident. The answers given by BERT were more unambiguous. Both models can be compared by analogy, in which Llama2, being a larger model, has a broader perspective, the same knowing more, but also knowing that it does not know everything (and could be more perfect). The bigger LLM eventually becomes a better tool in the context of expected epistemic uncertainty. It does a better job of distinguishing context within sentences in descriptions of software bugs. At the same time, it is more cautious. The perplexity curve for Llama2 in Figure 5.16 is slightly bent downward, suggesting the possibility of improving this result even further. Hence, removal of unique vocabulary is not related to catastrological drop in the quality of prediction.

The model not subjected to fine-tuning, but used with a simplified dataset,

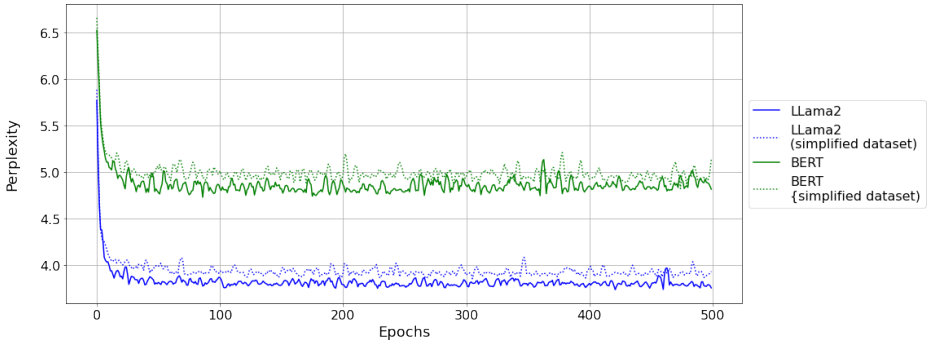


Figure 5.16: Preliminary training phase – changes in the perplexity metric

will achieve the same evaluation metrics values. Therefore, it seems that **as long as the surrounding context is comprehensible and sufficient, the LLM does not need the unique words to achieve the same quality of the entire classification model**. Llama2 gives more accurate answers and is less confident in their assignments.

Table 5.7: Results of preliminary LLM training for best epoch by highest accuracy and F1 score

Model	**BERT**		**LLama2**	
Dataset	original	simplified	original	simplified
Accuracy (best of 1)	42.89%	43.47%	54.50%	54.50%
Accuracy (best of 3)	74.54%	74.30%	81.86%	80.62%
F1 score	34.86%	34.29%	50.75%	50.45%
Percentiles mean and std P_1 - P_2	0.081 ± 0.038	0.083 ± 0.035	0.105 ± 0.039	0.117 ± 0.062
Perplexity	4.863	4.953	3.769	3.866
Epoch when achieved	29	29	112	330

5.4.2 A comparative analysis of modern architectures

The preliminary investigation with BERT and Llama2-7B provided foundational insights. It confirmed that larger models, while achieving higher accuracy, also exhibit greater caution in their predictions. Furthermore, it revealed that while unique, domain-specific vocabulary accelerates learning, its absence does not catastrophically degrade performance, as the models can leverage broader contextual understanding. These findings underscored the limitations of a surface-level approach and motivated a deeper, more systematic investigation. To build a truly reliable industrial system, it became clear that a more nuanced understanding of how different modern architectures interact with probabilistic methods was necessary. Therefore, the research proceeded to a comprehensive comparative analysis of four distinct LLMs, integrating a Bayesian framework to rigorously quantify and manage epistemic uncertainty.

Methodology: a two-stage approach

Stage 1: Representation learning and bayesian head This last layer is modeled as Bayesian logistic regression. For a given input embedding $\boldsymbol{x}_i$, which is the output of the frozen LLM encoder for the i-th data point, the log-odds of the positive class are given by a linear model:

$$\log \frac{p_i}{1-p_i} = \boldsymbol{\theta}^T \boldsymbol{x}_i \tag{5.1}$$

In this Bayesian formulation, the weight parameters $\boldsymbol{\theta}$ are not treated as a single point estimate, but as probabilistic variables. This allows us to capture the epistemic uncertainty of the model. The primary challenge in Bayesian inference is to calculate the true posterior distribution $P(\boldsymbol{\theta}|\mathcal{D})$, where $\mathcal{D}$ represents the training data. To accomplish this, a full posterior distribution for the weights of the classification layer is learned, improving the model uncertainty beyond a single estimate. The objective is to minimize the Kullback-Leibler (KL) divergence between our variational approximation $q(\boldsymbol{\theta})$ and the true posterior $P(\boldsymbol{\theta}|\mathcal{D})$:

$$q^*(\boldsymbol{\theta}) = \arg\min_q \mathrm{KL}\left(q(\boldsymbol{\theta}) \parallel P(\boldsymbol{\theta}|\mathcal{D})\right) \tag{5.2}$$

Since directly minimizing this KL divergence is difficult, because of the high-dimensional integral required to calculate the evidence term, its lower bound is maximized instead, known as the Evidence Lower Bound (ELBO). This objective function consists of two terms: the expected log-likelihood, which encourages the model to fit the data, and the KL divergence between the variational posterior and the prior, which acts as a regularization term. Maximizing the ELBO is equivalent to minimizing the KL divergence. In practice, the expectation within the ELBO is approximated using Monte Carlo sampling. Once the variational distribution $q^*(\boldsymbol{\theta})$ has been learned, predictions for a new input can be made by approximating the predictive posterior distribution. This is done by drawing N samples of the weights $\{\boldsymbol{\theta}^{(i)}\}_{i=1}^{N}$ from $q^*(\boldsymbol{\theta})$, calculating the output probability for each sample, and averaging the results:

$$\hat{p}_{\text{pred}} = \frac{1}{N} \sum_{i=1}^{N} S(\boldsymbol{\theta}^{(i)T} \boldsymbol{x}_{\text{new}}) \tag{5.3}$$

where $S(\cdot)$ is the sigmoid function. This averaging process integrates the learned uncertainty in the model weights, typically yielding more robust predictions.

Finally, the conceptual relationships between the variables in our model can be visualized using a plate diagram, as shown in Figure 5.17. The generative process is assumed for each of the N documents (bug reports), d_n, is composed of K tokens, t_k. These tokens are processed by the LLM (with fixed weights W) to produce embeddings, which are then classified by the final layer with latent weights θ. The model is trained to associate documents with one of C classes (software departments). To perform efficient and scalable probabilistic inference for complex models, VI is used to approximate intractable posterior distributions. For large datasets, traditional VI methods become computationally prohibitive, necessitating the use of Stochastic Variational Inference (SVI). In a discriminative setup, the feature extractor weights W are kept constant while SVI is used to learn the posterior distribution over the classification weights θ, based on the observed data. The hyperparameters α and β represent the priors on the document and class distributions, respectively.

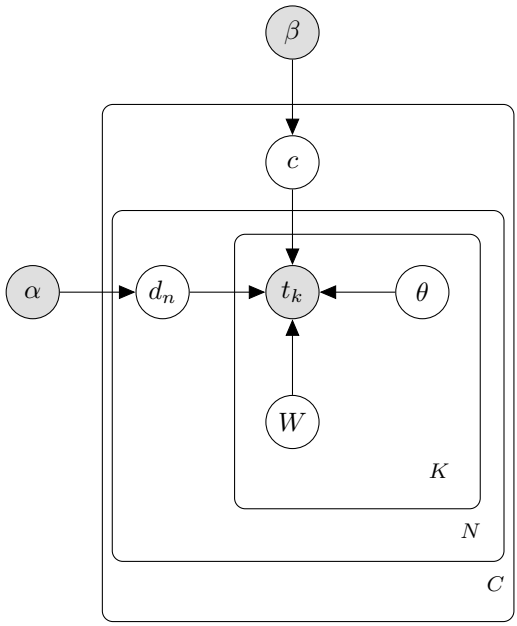


Figure 5.17: Probabilistic graphical model representation of the conceptual data relationships. N represents the number of documents, C is the number of classes, and K is the number of tokens. The weights of the LLM feature extractor, W, are treated as fixed, while a posterior distribution is learned over the classification head weights, θ. The observed variables are the tokens t_k and their corresponding class labels.

Stage 2: Overall pipeline The two-stage process is a methodological necessity, as including LoRA weights would be computationally impractical and methodologically flawed. The core issue lies in the fundamental incompatibility between the optimization dynamics of LoRA fine-tuning and variational inference. LoRA seeks a deterministic point estimate for its parameters by minimizing a standard loss function, whereas Stochastic Variational Inference (SVI) learns to shape the posterior approximation. Attempting to jointly optimize these contrasting objectives can introduce conflicting gradient signals, leading to optimization instability. Furthermore, the two procedures thrive under divergent hyperparameter regimes: LoRA fine-tuning typically necessitates a very low learning rate and a short training phase, whereas the convergence of a variational posterior often benefits from a longer training schedule and different learning rate dynamics. Therefore, decoupling these processes is essential for stable and effective training.

The initial phase involves the learning of representation using standard LoRA fine-tuning techniques. A conventional dense linear layer serves as the classification head, with training following a traditional point-wise optimization method. The objective is to efficiently adapt the LLM's extensive knowledge to the context of software bug reports, effectively teaching it the relevant language patterns without altering all its parameters. This results in a robust, task-optimized feature extractor. Once stage 1 ends, the LoRA adapters and base LLM weights are frozen. The deterministic classification head is discarded, replaced by a Bayesian layer. A second training phase begins, training only the parameters of this Bayesian head. Although Laplace-LoRA was proposed, this design is crucial as the aim is to **isolate Bayesian effects**. Freezing the feature extractor (the LLM with its LoRA adapters) neatly isolates the variable of interest: the Bayesian layer's behavior. This enables a focused study on how various priors influence the model's predictions and uncertainty estimates.

Experimental design

Our study involved four distinct open-source, small-scale (under 2 bilion parameters) LLMs as foundational architectures: Qwen3-0.6B [137], Llama-3.2-1B [50], Gemma-3-1b-it [118], and Deepseek-Coder-1.3b [52]. Using these smaller yet effective LLMs is a deliberate choice to enhance usability and accessibility, allowing

for a broader adoption among researchers and practitioners, even with limited resources.

Prior distributions The Bayesian classifier is set up to evaluate our hypotheses with three distinct priors, selected for their specific regularization characteristics. **Normal** (Gaussian), a standard L_2-like regularization and non-sparsity-inducing prior defined by its mean ($\mu = 0$) and standard deviation (σ). **Laplace**, a sparsity-inducing prior that promotes weights to be exactly zero, effectively performing regularization and feature selection like L_1 within the classification layer. It is defined by its location ($\mu = 0$) and the scale parameter (σ). Each weight and bias requires two variational parameters, as defined in Equation (5.4). Thus, the total parameter count is effectively doubled when compared to a non-Bayesian model.

$$\theta_{\mathrm{L}} = (\mathrm{num\_classes} \times \mathrm{in\_features} \times 2) + (\mathrm{num\_classes} \times 2) \tag{5.4}$$

The term **horseshoe prior** stands for a hierarchical distribution designed to induce sparsity. This allows the model to learn the appropriate level of regularization for each individual weight. The guide for a horseshoe prior requires additional variational parameters for the local (λ) and global (τ) scale parameters that define the prior. Each of these scales is often modeled with a Half-Cauchy distribution, which has its own scale parameter to be optimized, as defined in Equation (5.5).

$$\theta_{\mathrm{horsheshoe}} = \theta_{\mathrm{L}} + (\underbrace{1}_{\text{global } \tau_w} + \underbrace{1}_{\text{global } \tau_b} + \underbrace{\mathrm{num\_classes} \times \mathrm{in\_features}}_{\text{local } \lambda_w} + \underbrace{\mathrm{num\_classes}}_{\text{local } \lambda_b}) \tag{5.5}$$

The Table 5.8 summarizes the number of parameters for each of the specified models, assuming a classification task with $|C| = 8$ classes.

Hyperparameter tuning To assess how the narrowness in the prior, and consequently the constraints, influence the models' performance, the following set of parameters σ **values** are utilized: $\{0.01, 0.1, 0.5, 1.0, 2.0\}$. Subsequently, to ex-

Table 5.8: Total number of variational parameters for different LLM feature vectors

Model Name	Feature Vector Size	Prior	Variational Parameters
Qwen3	1024	normal / Laplace	16,400
		horseshoe	24,602
Gemma-3	1152	normal / Laplace	18,448
		horseshoe	27,674
Deepseek-Coder	2048	normal / Laplace	32,784
		horseshoe	49,178
Llama3.2	2048	normal / Laplace	32,784
		horseshoe	49,178

amine how the number of samples impacts the uncertainty estimation, the set of **number of MC samples** chosen is: $\{16, 32, 64, 128, 256\}$.

Architectural considerations and hypotheses Each LLM is associated with distinct architectural differences, enabling us to formulate preliminary expectations regarding their compatibility and potential application with various priors.

- **Attention mechanisms**:

  Deepseek-Coder uses standard Multi-Head Attention (MHA). It is the original attention mechanism introduced in the Transformer architecture.

  Qwen3 and Llama-3.2 adopt the memory-saving Grouped-Query Attention (GQA). It provides a compromise between performance and efficiency by combining several query heads to use one key and value head. This greatly reduces memory usage during inference compared to MHA, resulting in faster and less resource-intensive operations. From a Bayesian perspective, this architecture offers a stable feature space likely to work effectively with any prior. It avoids the high redundancy of MHA and the bottleneck issue in Multi-Query Attention (MQA).

  The more aggressively optimized Gemma-3 uses MQA. It improves perfor-

mance by using a single shared key and value head across all query heads. This significantly reduces memory usage and boosts inference speed, but may result in a drop in quality compared to GQA or MHA due to lower representational capacity. Using a strongly sparsity-inducing prior, such as Laplace, might be harmful by potentially nullifying crucial weights in the already constrained representation. In contrast, the horseshoe prior may be capable of preserving weights for important signals, which may be essential to exceed the MQA limitation and achieve best performance.

- **Normalization and activation functions**:

  Models, such as Llama-3.2, Deepseek-Coder, and Gemma-3, prefer RMS normalization (RMSNorm). It trims down Layer normalization by normalizing only the variance, bypassing re-centering, thus speeding up computation. Conversely, Qwen3 uses traditional Layer normalization, normalizing inputs across all features for stable training dynamics. A well-normalized model offers a cleaner, more Gaussian-like feature distribution in the final layer, suggesting that normalization methods will similarly impact all prior distributions.

  The models are split by activation functions. Llama-3.2 and Deepseek-Coder use SwiGLU, whereas Qwen3 and Gemma-3 use GeGLU. Both are Gated Linear Unit (GLU) variants that uses gating mechanisms to manage data flow and create highly nonlinear, data-dependent features. This means that for any given input, many neurons might be inactive. Theoretically, this native data-driven sparsity makes the architecture a natural fit for a horseshoe prior, which is designed to model signals that are sparse but contain a few large effects, mirroring how these gates might allow strong signals to pass through only specific pathways.

- **Other features**:

  The Qwen3 is unique in its use of **parameter linking** (also known as weight tying) between the embedding and final layers. This technique forces the two layers to share the same weight matrix, which reduces the total number of model parameters. This acts as a form of built-in regularization,

which can help prevent overfitting and improve generalization, especially in smaller models. The normal prior may lead to a doubly regularized model, likely making it very robust against overconfidence but at risk of underfitting. A Laplace prior would create an interesting interaction, that may enforce sparsity on a layer that is also tied to the input embeddings, effectively pruning the vocabulary's importance for the task. Then, a horseshoe prior is theoretically the most powerful match, as it can expertly navigate the complex, constrained weight space, identifying a sparse set of strong features.

Evaluation metrics Classification accuracy is evaluated using standard accuracy metrics, accuracy for top-N predictions where $N = 3$, and F1 score metrics. In the NLP domain, PPL is calculated on the basis of class assignments in the epoch. In addition, the calibration of the model is assessed using Expected Calibration Error (ECE). It quantifies the difference between the predicted confidence of the model and its actual accuracy across different confidence bins, with a lower ECE indicating a better calibration.

For each input, a prediction distribution is generated by sampling weights $\{\boldsymbol{\theta}^{(i)}\}_{i=1}^{N}$ from the posterior $q^*(\boldsymbol{\theta})$. The process involves generating a range of possible predictions for each input by sampling different model weights. Several **percentiles** are identified within this prediction range: $P_{2.5}$, P_5, P_{25}, P_{75}, P_{95}, and $P_{97.5}$. These percentiles are used to define three different CI for each class, which represent the range where the true probability is likely to fall with a certain confidence:

- **95% CI**: This is the widest range, from $P_{2.5}$ to $P_{97.5}$.
- **90% CI**: This is a slightly narrower range, from P_5 to P_{95}.
- **50% CI**: This is the narrowest range, from P_{25} to P_{75}.

Results and analysis

Impact of prior distributions and architectural choices Instead of simply assessing a model's natural ability to spot unanswerable questions, the development of

this skill is actively pursued using a Bayesian framework, which lets the model quantitatively express uncertainty through the width of CI. Past studies have shown that although fine-tuning and in-context learning are efficient for adaptation to new domains, they can surprisingly impair model calibration, resulting in overly confident predictions, particularly in low-shot scenarios [144]. Our study addresses this problem by showing that a post-hoc Bayesian layer can mitigate these effects. The notable decrease in ECE in our experiments (Table 5.9) confirms that this architectural adjustment effectively restores the reliability that is often diminished during specialization.

A recent method, Laplace-LoRA, directly applies a Laplace approximation to all LoRA parameters for better calibration [138]. While it gives a comprehensive Bayesian analysis of adapted weights, our approach intentionally isolates the Bayesian inference to the final classification head. This design is based on the idea that the critical epistemic uncertainty in classification is in the decision-making layer. The strong performance of Qwen3 and Llama-3.2 models supports this. It highlights uncertainty quantification where it matters most for classification, complementing the effectiveness of Bayesian fine-tuning [138]

Now, the following research questions are formulated as follows:

RQ5: Impact of prior distribution characteristics Across the four models and for both normal and Laplace priors, there is a clear and consistent trend: increasing the prior scale σ from 0.01 to 2.0 systematically improves all key metrics. This demonstrates that for high-capacity, pre-trained LLM as feature extractors, **over-regularization is a much greater risk than under-regularization**. When σ is very small, such as 0.01, it enforces a strong L_1 or L_2 regularization that significantly drives the posterior weights closer to zero. This results in severe underfitting, evidenced by low F1 scores and high ECE, due to inaccurate and poorly calibrated model predictions. For example, in the Llama-3.2 model using a normal prior, increasing σ from 0.01 to 0.10 leads to a substantial improvement in the F1 score, elevating it from a poor 0.477 to a more satisfactory 0.729, while the ECE decreases from 0.288 to 0.230. As σ increases towards 1.0 and 2.0, the prior loses its informativeness, becoming "flatter", which gives the model the flexibility to capitalize on the rich and dense features provided by the LLM. Consequently,

this significantly improves both accuracy and calibration, and all models reach peak performance when $\sigma \geq 1.0$. The interaction between σ and the CI width offers valuable information. Interestingly, for various setups, the CI width does not always expand with σ. For example, with Qwen3 (Laplace), the width of 95% CI decreases from 0.453 at $\sigma = 0.01$, reaching its lowest point at 0.223 when $\sigma = 2.0$. This suggests that as the model becomes more accurate due to the loosened prior, its confidence in correct predictions appropriately rises, leading to narrower, more informative credible intervals. The ideal configuration balances being neither too restricted to cause underfitting nor too permissive to promote overconfidence. For this particular task, adopting a weak prior ($\sigma = 2.0$) seems to achieve this balance effectively.

RQ6: Effectiveness of different prior distributions Contrary to the theoretical appeal of the horseshoe prior, it consistently and significantly underperforms compared to the simpler normal and Laplace priors. Figure 5.18 shows the average training results for the F1 score (blue) and ECE (orange). Further calibration optimization significantly reduced classification performance. Contrary to the models with the best performance, Llama-3.2 and Qwen3, the F1 score obtained using the horseshoe prior is significantly lower (0.769 and 0.774, respectively) than the highest scores obtained with the Laplace or normal priors, which exceed 0.800.

This indicates two potential concerns. Firstly, the additional complexity of the horseshoe prior and the varying parameters could have impeded the successful optimization through SVI, possibly causing convergence at an unfavorable local optimum. Secondly, and more fundamentally, the dense, high-dimensional embeddings generated by the final layer of a refined LLM may not align with the horseshoe's "global-local" sparsity approach. The strength of the horseshoe lies in pinpointing a few significant effects among numerous that are zero-valued. Yet, the LLM embeddings likely consist of numerous small but collectively important features. The horseshoe prior might misidentify and suppress these crucial features, resulting in information loss and compromising performance.

In the comparison between normal and Laplace priors, there is no universal winner, but the **Laplace prior frequently provides a slight advantage, especially**

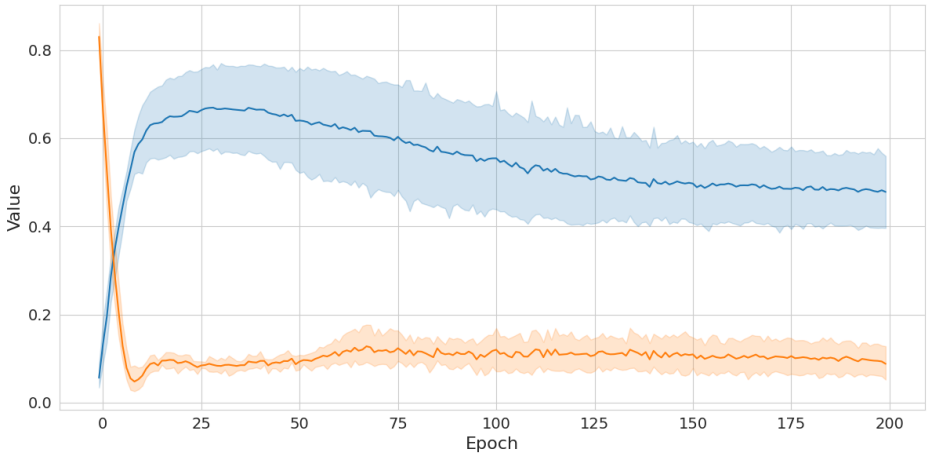


Figure 5.18: The mean validation outcomes for models using the horseshoe prior. The blue line denotes F1, while the orange represents ECE. F1 shows visible saturation followed by a decrease as better calibration is pursued

in terms of calibration. For the high-performing Qwen3 model, the F1 score is 0.806 for both priors when $\sigma = 2.0$, demonstrating parity. Yet, the Laplace prior demonstrates an exceptional ECE of 0.015, superior to the normal prior's 0.024. This indicates that the L1-style regularization characteristic of the Laplace prior is somewhat more successful in eliminating redundant features from the final embedding, yet being less aggresive than horseshoe.

RQ7: Interaction with LLM architecture Selecting the foundational LLM is crucial as it largely determines the upper limit of the final performance, a ceiling that the Bayesian layer cannot exceed. The models can be categorized into two clear performance tiers:

- **Tier 1 (Superior Performance):** Llama-3.2 and Qwen3 consistently achieve F1 scores exceeding 0.80.
- **Tier 2 (Moderate Performance):** Gemma-3 and DeepSeek-Coder stabilize around an F1 score of approximately 0.65.

This performance gap is directly tied can be attributed to differences in ar-

chitecture. In particular, the two leading models, Llama-3.2 and Qwen3, employ GQA. This strongly supports the theory that GQA offers an improved balance between representational capacity and efficiency, resulting in improved and more consistent feature embeddings essential for the classification task. On the other hand, Gemma-3's implementation of MQA seems to impose a representational bottleneck. Its lower performance supports the idea that the extensive parameter sharing inherent in MQA might cause a quality loss that the Bayesian head struggles to compensate for. More unexpectedly, the subpar performance of DeepSeek-Coder, which employs standard MHA, is noteworthy. Despite being theoretically the most expressive, MHA's significant redundancy potentially results in feature embeddings that become noisier and less capable of generalization post fine-tuning, rendering them more challenging for the Bayesian layer to model effectively.

Among the notable discoveries is the effectiveness of **Qwen3**, which integrates GQA and uses **parameter linking** (weight tying) between its embedding and classifier layers. This built-in regularization, paired with the L1-like sparsity introduced by the Laplace prior, cultivates a multiply regularized model that stands out as the best calibrated among all evaluated. This collaborative mechanism underscores the significant influence that deeply embedded architectural decisions within the model can have on overall uncertainty quantification. Interestingly, neither the size of the feature vector nor the choice between SwiGLU and GeGLU activation functions were key factors, as models with varying performances had different configurations. Qwen3, despite its smaller feature vector, performs on par with Llama, suggesting a promising path for optimizing LLM size and complexity.

Industrial application

RQ8: Reliability of the system A reliable system is characterized not only by its precision, but also by its well-calibrated nature and provision of informative uncertainty estimates. The analysis identifies that the most effective model for establishing a such system is **Qwen3 with a Laplace prior and** $\sigma = 2.0$. This model achieves a considerable F1 score of 0.806, an exceptionally low ECE of 0.015, which is the best among all configurations evaluated, and a relatively nar-

row 95% CI width of 0.223. Therefore, an engineer can potentially have more confidence in its predictions and, importantly, in its articulation of uncertainty. The Llama-3.2 configuration with a Laplace prior and $\sigma = 2.0$ is a close contender, offering an insignificantly higher accuracy (F1 0.807) at the expense of a moderately less impressive calibration (ECE 0.043), although still commendable. In sharp contrast stands the **DeepSeek-Coder with a horseshoe prior** configuration, which represents a dangerously "confidently wrong" model. It has an exceedingly narrow width CI, indicating high confidence; however, its F1 score struggled to a poor 0.593, while its ECE is substantially elevated at 0.109. Such a model is more detrimental than beneficial in an operational setting, as it could fail without alerting the user, leading to many misclassifications.

A review of the validation results in Table 5.9 and the production results in Tables 5.10-5.13 reveals a clear hierarchy for industrial deployment. The **Qwen3** and **Llama-3.2** models, particularly with a relaxed Laplace prior ($\sigma = 2.0$), form the best option for a reliable Human-In-The-Loop (HITL) system. They consistently provide the best combination of high F1 scores (more than 0.80), excellent calibration (ECE as low as 0.015 for Qwen3), and a significant automation potential (more than 30% of tickets). In contrast, **Gemma-3** and **DeepSeek-Coder** constitute a second tier; while functional, their lower accuracy and significantly reduced automation rates make them less attractive for deployment.

Most importantly, the results serve as a warning against the use of the **horseshoe prior to** in this application. Across the four foundational models, the horseshoe configuration produced dangerously over-confident and inaccurate models. For example, Table 5.10 shows that it confidently processes 87% of tickets while being wrong on more than one in five of them – a catastrophic failure rate for a critical system. This demonstrates that theoretical elegance does not guarantee practical utility and that simpler priors provided a much more robust and reliable solution.

Production testing: generalization, uncertainty, and humain-in-the-loop implications

This section examines the practical use of trained models in a industrial production setting, which extends beyond typical evaluation metrics. It evaluates model

performance on new, unseen data and emphasize the role of uncertainty estimates in a HITL system. Particularly crucial in high-stakes fields like telecommunications, this workflow aims to complement human expertise by fully automating tasks when system confidence is high. The subsequent Tables 5.10 to 5.13 present the best model configurations, analyzing the balance between automation rates, accuracy, and the potential computional requirements of uncertainty estimation (measured by Monte Carlo sample numbers).

Role of Monte Carlo samples Using more MC samples to approximate the posterior affects both the dependability and cost of predictions. Increasing the number of samples from 16 to 64 significantly improves uncertainty estimates, increases precision for confident predictions, and improves the F1 score for less certain ones (for example, Qwen3 Laplace 95% CI sees an increase from 0.62 to 0.64), while reducing PPL. This indicates a more stable posterior approximation. However, moving from 128 to 256 samples brings minimal gains at double the inference cost. In particular, more samples might slightly *decrease* confidence predictions, demonstrating a well-calibrated model's ability to flag uncertain cases more accurately. In production environments weighing reliability and speed, **64 to 128 samples are preffered**.

The automation potential The "Confident (%)" column reflects automation potential, while the F1 of confident versus non-confident predictions (in parentheses) indicates the automation quality and the effectiveness of the model for uncertain cases. Models like **Llama-3.2** and **Qwen3**, using a relaxed Laplace or normal prior, demonstrate strong HITL capabilities. Qwen-3 (Laplace, 128 MC samples, 95% confidence), for instance, can independently resolve 36.60% of incoming tickets with a 98% F1 score. This greatly improves engineering efficiency. The remaining 63.40% are escalated to an engineer, maintaining a 0.67 F1 score. In contrast, **Gemma-3** and **Deepseek-Coder** are more conservative. Under similar settings (Laplace, 128 MC, 95% CI), Deepseek-Coder is confident on just 8.2% of data. Despite the high accuracy in confident predictions (98%), its automation rate below 10% offers a low industrial effectiveness. Deployment and maintenance might not be justified.

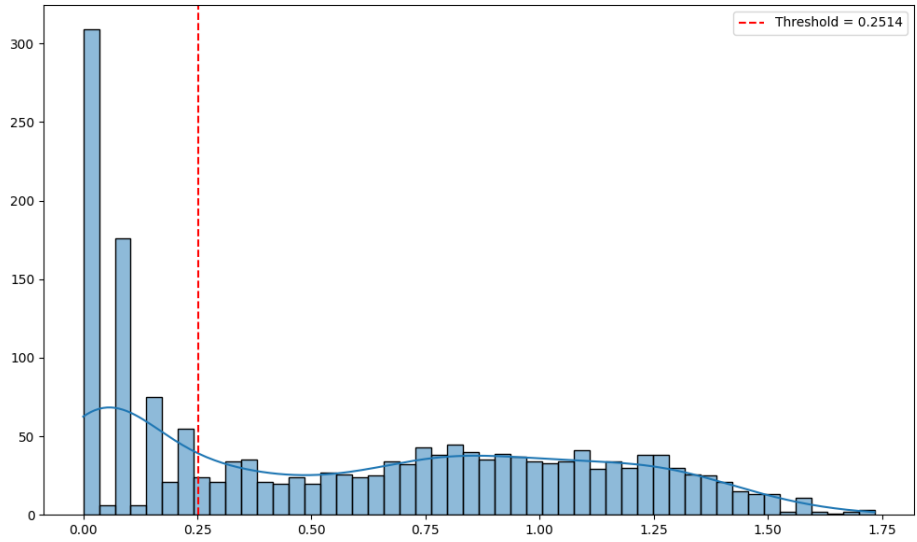


Figure 5.19: Illustration of Qwen3 uncertainty distribution for production samples. The y-axis denotes ticket count, while the x-axis shows their calculated CI width. The CI threshold indicating prediction confidence is determined during training (in this case normal distribution, CI 95% with $\sigma = 1.0$)

Production testing data indicates that systems with the horseshoe prior are not only suboptimal but also severely overconfident and unreliable. It presents a critical and negative outcome. For example, Qwen3 with the horseshoe prior confidently handles 87% of tickets but only reaches 79% precision, leading to more than one in five critical bug reports being misrouted. The situation worsens with Deepseek-Coder, where confidence stands at 85% with an accuracy of about 0.51, resulting in errors almost half the time. Then, the PPL scores for horseshoe configurations are far greater than those for normal / Laplace (e.g., > 3000 for the Deepseek-Coder horseshoe vs. <6 for Laplace), indicating a modeling failure. This clearly answers RQ2: the theoretical appeal of the horseshoe prior results in a practically unusable model.

Our analysis goes beyond standard metrics by emphasizing Bayesian predictive uncertainty, crucial for preventing errors with Out of Distribution (OOD) data. Designed to recognize challenging OOD samples for typical models, it identifies errors in rushed inputs, often full of abbreviations, typos, and indirect hints. Instead of guessing, the Bayesian classification layer assigns a high uncertainty, acknowledging that missing context impedes confidence. It also detects implicit contexts, such as ambiguous references, resulting in broad weight distributions and signal uncertainty. Vague queries, such as "This could be related to ticket #12345" are flagged due to insufficient detail. This approach performs exceptionally well with new, unseen situations by identifying when input features do not match data from a trianing phase. The model response is a widened predictive distribution that indicates extrapolation, not interpolation. The strength of the system lies in turning uncertainty into opportunities for human intervention.

5.5 Scientific and industrial impact

In a typical situation, research reported in literature presents new or old but modified neural network architectures that were examined on a typical set of metrics, which is classification accuracy, usually producing high results. The main limitation is that several state-of-the-art methods are neither designed for use in industrial pipelines nor considered for integration in demanding environments. These models are prepared and then evaluated on open and well-formed

datasets, which are significantly different from those available in closed technological processes. For example, BERT was trained on different datasets to perform different tasks, for example, text classification with DBpedia by [9] or sentence classification with SciCite by [30], question answering with SQuAD by [101] or CoQA by [102]. [13] introduced the SciBERT, which has been prepared to work in environments closer to science and technology. Unfortunately, it is trained on papers from the corpus of `semanticscholar.org`, which includes many different scientific fields, while the most appropriate would be a corpus focused on telecommunications. This architecture was evaluated on a general dataset of ACL-ARC by [15] or ChemProt by [96]. The Llama2 training corpus is based on open sourced texts, articles and repositories from Wikipedia, ArXiv or GitHub. This raises a general problem of the company, when decision on possible fine-tuning of the LLM must be made. This leads to the situation where multiple tuned models are made to be specialists in niche applications, making them expensive to maintain. Therefore, an alternative point of view is suggested that aims not only to mitigate the problem of uncertainty related to exclusive content of the data, but also to limit the costs of deployment of the LLM models in the industrial scenario.

From a machine learning point of view, we made the following contribution – we evaluated the use of not fine-tuned LLM in Bayesian scenario, when working with a highly exclusive dataset. We discovered that a large enough LLM does not need to know all the unique vocabulary. As long as the description of the software bug is extensive and the context is easy to understand, the model is able to work effectively with such text. Therefore, a potential research direction should be directed toward understanding what language and text properties affect prediction quality. It would be necessary to see if certain linguistic constructions can introduce additional uncertainty (e.g., multiple complex sentences, those with multiple verbs referring to many different or consecutive actions).

From a business and software quality assurance point of view, the economic aspects of LLM implementation must be taken into account. The Bayesian approach provides natural safety against the over-fitting problem that is expected when using such a powerful model as encoder on a small corporate and inclusive dataset. Additionally, the system leads the user to validate the confidence of

the answer based on the given input data. Such a system then becomes a closed loop in which the system signals a problem with understanding of the content and gives the engineer the opportunity to add depth to selected parts of the described problem. In addition, this allows the fine-tuning phase to be skipped, reducing the cost of maintaining and using such a model for longer time.

It is important to highlight that the proposed methodology is not restricted to software quality assurance within the telecommunications sector. This approach can also be applied to various other types of problems. The same LLM can be used for several different applications, such as a chatbot for engineers or in the customer service area.

Table 5.9: Performance and epistemic uncertainty metrics across different prior configurations. For each model, the row with the highest F1 score and smallest ECE is in **bold**

Model	Prior	σ	Acc.	Top-3	F1	PPL	ECE	CI (P_1 - P_0)		
			%	%	%			95%	90%	50%
Gemma-3	horseshoe		0.629	0.879	0.589	5.985	0.089	0.312	0.263	0.102
	Laplace	0.01	0.444	0.779	0.395	7.388	0.198	0.564	0.476	0.131
		0.10	0.588	0.854	0.540	6.260	0.112	0.482	0.398	0.124
		0.50	0.657	0.886	0.624	5.739	0.066	0.472	0.398	0.131
		1.00	0.666	0.893	0.639	5.699	0.076	0.449	0.370	0.119
		2.00	0.669	0.897	0.643	5.589	0.043	0.257	0.217	0.087
	normal	0.01	0.483	0.762	0.393	7.395	0.239	0.571	0.482	0.142
		0.10	0.586	0.854	0.526	6.345	0.120	0.508	0.424	0.133
		0.50	0.650	0.888	0.616	5.847	0.086	0.464	0.386	0.122
		1.00	0.668	0.897	0.641	5.661	0.064	0.440	0.365	0.114
		2.00	**0.672**	**0.903**	**0.648**	**5.553**	**0.042**	**0.432**	**0.335**	**0.100**
Llama-3.2	horseshoe		0.775	0.955	0.769	5.131	0.118	0.416	0.350	0.131
	Laplace	0.01	0.536	0.794	0.496	7.228	0.270	0.532	0.438	0.111
		0.10	0.748	0.938	0.741	5.601	0.202	0.349	0.280	0.093
		0.50	0.798	0.958	0.794	4.901	0.087	0.295	0.236	0.082
		1.00	0.809	0.966	0.806	4.748	0.053	0.282	0.226	0.078
		2.00	**0.809**	**0.966**	**0.807**	**4.710**	**0.043**	**0.352**	**0.298**	**0.116**
	normal	0.01	0.536	0.769	0.477	7.388	0.288	0.586	0.480	0.109
		0.10	0.739	0.931	0.729	5.765	0.230	0.380	0.302	0.096
		0.50	0.794	0.960	0.789	4.977	0.103	0.318	0.253	0.087
		1.00	0.804	0.966	0.801	4.797	0.066	0.278	0.224	0.079
		2.00	0.808	0.972	0.805	4.697	0.040	0.372	0.292	0.096
Qwen3	horseshoe		0.779	0.952	0.774	5.003	0.083	0.262	0.222	0.089
	Laplace	0.01	0.580	0.841	0.488	6.917	0.280	0.453	0.359	0.107
		0.10	0.767	0.941	0.760	5.315	0.157	0.312	0.248	0.085
		0.50	0.798	0.965	0.795	4.802	0.056	0.269	0.217	0.078
		1.00	0.804	0.968	0.802	4.700	0.031	0.243	0.197	0.071
		2.00	**0.808**	**0.970**	**0.806**	**4.620**	**0.015**	**0.223**	**0.190**	**0.077**
	normal	0.01	0.567	0.775	0.462	7.197	0.310	0.494	0.390	0.109
		0.10	0.754	0.942	0.743	5.468	0.178	0.325	0.258	0.087
		0.50	0.797	0.962	0.793	4.844	0.065	0.281	0.226	0.081
		1.00	0.803	0.964	0.800	4.737	0.043	0.251	0.205	0.075
		2.00	0.808	0.968	0.806	4.646	0.024	0.544	0.446	0.135
DeepSeek-Coder	horseshoe		0.633	0.883	0.593	5.965	0.109	0.080	0.069	0.029
	Laplace	0.01	0.541	0.804	0.470	7.918	0.401	0.455	0.381	0.130
		0.10	0.617	0.900	0.577	6.339	0.184	0.440	0.367	0.135
		0.50	0.661	0.910	0.633	5.712	0.083	0.419	0.347	0.129
		1.00	0.667	0.910	0.640	5.576	0.051	0.401	0.334	0.131
		2.00	**0.675**	**0.923**	**0.654**	**5.509**	**0.045**	**0.063**	**0.055**	**0.023**
	normal	0.01	0.500	0.786	0.462	7.950	0.366	0.528	0.492	0.134
		0.10	0.617	0.884	0.566	6.521	0.217	0.428	0.359	0.129
		0.50	0.653	0.910	0.621	5.768	0.082	0.459	0.381	0.134
		1.00	0.666	0.914	0.640	5.621	0.065	0.433	0.358	0.136
		2.00	0.673	0.915	0.650	5.509	0.038	0.403	0.338	0.128

Table 5.10: **Generalization and usability metrics for Qwen3 with varying MC samples**. The table shows performance for the best-known configurations (Laplace and normal priors with $\sigma = 2.0$) and horseshoe prior for comparison. Values in parentheses represent metrics for non-confident predictions

MC Samples	Confident (%)			Accuracy			F1			PPL		
	95%	90%	50%	95%	90%	50%	95%	90%	50%	95%	90%	50%
Laplace												
16	43.00	33.90	33.40	0.97 (0.62)	0.98 (0.66)	0.98 (0.67)	0.97 (0.62)	0.98 (0.66)	0.98 (0.66)	1.91 (4.94)	1.65 (4.66)	1.60 (4.70)
32	37.60	35.30	23.90	0.97 (0.65)	0.98 (0.66)	0.98 (0.70)	0.97 (0.64)	0.97 (0.66)	0.98 (0.70)	1.58 (3.30)	1.56 (3.24)	1.55 (2.91)
64	39.00	36.10	19.40	0.98 (0.65)	0.98 (0.66)	0.98 (0.73)	0.98 (0.64)	0.98 (0.66)	0.98 (0.73)	1.39 (2.87)	1.42 (2.75)	1.42 (2.39)
128	36.60	33.80	20.90	0.98 (0.66)	0.99 (0.67)	1.00 (0.72)	0.98 (0.66)	0.98 (0.67)	0.99 (0.72)	1.24 (2.60)	1.21 (2.56)	1.14 (2.30)
256	35.90	32.90	19.10	0.98 (0.67)	0.98 (0.68)	0.99 (0.73)	0.98 (0.66)	0.98 (0.68)	0.99 (0.73)	1.14 (2.49)	1.14 (2.40)	1.11 (2.14)
normal												
16	43.00	31.80	31.40	0.96 (0.61)	0.97 (0.66)	0.97 (0.66)	0.96 (0.61)	0.97 (0.66)	0.97 (0.66)	1.99 (4.50)	1.83 (4.09)	1.84 (4.06)
32	36.90	35.20	24.60	0.97 (0.66)	0.98 (0.66)	0.99 (0.70)	0.97 (0.66)	0.98 (0.66)	0.99 (0.70)	1.56 (3.37)	1.40 (3.50)	1.30 (3.16)
64	35.60	32.00	15.80	0.98 (0.66)	0.98 (0.67)	0.99 (0.73)	0.98 (0.65)	0.98 (0.67)	0.98 (0.73)	1.36 (2.60)	1.38 (2.49)	1.31 (2.25)
128	35.70	32.60	18.70	0.98 (0.66)	0.98 (0.67)	0.99 (0.72)	0.98 (0.66)	0.98 (0.67)	0.99 (0.73)	1.16 (2.62)	1.17 (2.52)	1.12 (2.23)
256	34.30	31.70	18.40	0.98 (0.67)	0.99 (0.68)	1.00 (0.73)	0.98 (0.67)	0.98 (0.68)	1.00 (0.73)	1.12 (2.44)	1.09 (2.39)	1.02 (2.14)
horseshoe												
16	88.00	86.00	81.40	0.79 (0.36)	0.80 (0.37)	0.81 (0.41)	0.79 (0.38)	0.80 (0.39)	0.81 (0.42)	37.65 (41.64)	35.81 (55.76)	35.83 (49.95)
32	87.40	85.70	80.60	0.79 (0.36)	0.80 (0.37)	0.82 (0.41)	0.79 (0.38)	0.80 (0.39)	0.82 (0.43)	30.72 (51.00)	30.59 (49.17)	30.93 (41.50)
64	87.80	86.00	80.10	0.79 (0.37)	0.80 (0.36)	0.82 (0.43)	0.79 (0.38)	0.80 (0.37)	0.82 (0.44)	28.47 (57.56)	28.15 (56.32)	29.75 (36.77)
128	87.80	85.50	79.50	0.79 (0.35)	0.80 (0.35)	0.82 (0.42)	0.79 (0.36)	0.80 (0.36)	0.82 (0.44)	27.23 (44.42)	26.13 (52.50)	27.45 (35.33)
256	87.30	85.50	79.50	0.79 (0.34)	0.80 (0.36)	0.82 (0.40)	0.79 (0.36)	0.80 (0.38)	0.82 (0.42)	24.68 (41.10)	24.17 (43.49)	24.55 (34.49)

Table 5.11: **Generalization and usability metrics for Deepseek-Coder with varying MC samples**. The table shows performance for the best-known configurations (Laplace and normal priors with $\sigma = 2.0$) and horseshoe prior for comparison. Values in parentheses represent metrics for non-confident predictions

MC Samples	Confident (%)			Accuracy			F1			PPL		
	95%	90%	50%	95%	90%	50%	95%	90%	50%	95%	90%	50%
Laplace												
16	11.20	10.00	6.30	0.94 (0.51)	0.96 (0.52)	0.96 (0.53)	0.92 (0.50)	0.95 (0.50)	0.95 (0.52)	2.16 (5.09)	1.87 (5.11)	2.29 (4.85)
32	10.00	9.00	3.50	0.98 (0.54)	0.98 (0.54)	1.00 (0.57)	0.98 (0.52)	0.97 (0.53)	1.00 (0.55)	1.29 (4.20)	1.32 (4.14)	1.00 (3.92)
64	8.20	7.50	3.60	0.98 (0.53)	0.98 (0.54)	0.98 (0.55)	0.97 (0.52)	0.97 (0.52)	0.98 (0.54)	1.45 (3.53)	1.50 (3.50)	1.46 (3.39)
128	8.20	8.00	4.10	0.98 (0.53)	0.98 (0.54)	0.99 (0.55)	0.97 (0.52)	0.97 (0.52)	0.98 (0.53)	1.12 (3.21)	1.13 (3.21)	1.08 (3.08)
256	8.20	7.80	3.30	0.98 (0.53)	0.98 (0.53)	1.00 (0.55)	0.97 (0.51)	0.97 (0.51)	1.00 (0.53)	1.15 (3.32)	1.15 (3.30)	1.02 (3.16)
normal												
16	11.00	9.20	5.00	0.96 (0.50)	0.97 (0.51)	0.99 (0.53)	0.96 (0.48)	0.96 (0.49)	0.98 (0.51)	1.76 (5.72)	1.64 (5.62)	1.30 (5.39)
32	8.60	7.20	1.90	0.97 (0.52)	0.97 (0.53)	1.00 (0.55)	0.96 (0.50)	0.96 (0.51)	1.00 (0.53)	1.59 (3.91)	1.71 (3.84)	1.00 (3.71)
64	8.20	7.80	4.10	0.97 (0.53)	0.98 (0.53)	1.00 (0.54)	0.96 (0.51)	0.97 (0.51)	1.00 (0.52)	1.28 (3.66)	1.27 (3.65)	1.01 (3.54)
128	7.90	7.50	3.00	0.98 (0.52)	0.99 (0.52)	0.98 (0.55)	0.97 (0.50)	0.98 (0.50)	0.97 (0.52)	1.21 (3.26)	1.18 (3.25)	1.33 (3.09)
256	8.10	7.70	2.60	0.98 (0.52)	0.98 (0.52)	1.00 (0.55)	0.97 (0.50)	0.97 (0.50)	1.00 (0.52)	1.15 (3.28)	1.16 (3.27)	1.02 (3.10)
horseshoe												
16	85.80	82.70	70.10	0.51 (0.36)	0.51 (0.37)	0.53 (0.36)	0.42 (0.39)	0.41 (0.40)	0.43 (0.37)	3415.11 (28.55)	4102.55 (28.81)	6955.14 (69.11)
32	86.00	83.00	68.80	0.51 (0.37)	0.52 (0.36)	0.54 (0.37)	0.42 (0.41)	0.42 (0.40)	0.43 (0.39)	2750.41 (24.99)	3160.11 (30.01)	6510.92 (52.55)
64	85.20	82.30	68.60	0.51 (0.38)	0.52 (0.38)	0.54 (0.38)	0.42 (0.41)	0.42 (0.40)	0.42 (0.40)	2250.93 (19.88)	2580.19 (23.51)	4950.22 (45.94)
128	85.00	82.10	68.70	0.51 (0.36)	0.52 (0.35)	0.55 (0.35)	0.42 (0.39)	0.42 (0.38)	0.43 (0.38)	1980.52 (16.94)	2250.81 (19.22)	4010.53 (41.83)
256	84.90	81.80	68.10	0.51 (0.37)	0.52 (0.36)	0.55 (0.36)	0.42 (0.40)	0.42 (0.39)	0.43 (0.38)	1690.17 (15.99)	1810.23 (24.81)	3390.41 (42.05)

Table 5.12: **Generalization and usability metrics for Llama-3.2 with varying MC samples**. The table shows performance for the best-known configurations (Laplace and normal priors with $\sigma = 2.0$) and horseshoe prior for comparison. Values in parentheses represent metrics for non-confident predictions

MC Samples	Confident (%)			Accuracy			F1			PPL		
	95%	90%	50%	95%	90%	50%	95%	90%	50%	95%	90%	50%
Laplace												
16	37.90	28.80	27.70	0.96 (0.62)	0.97 (0.66)	0.98 (0.66)	0.96 (0.62)	0.97 (0.66)	0.97 (0.66)	1.97 (4.58)	1.68 (4.39)	1.63 (4.37)
32	33.60	27.20	18.10	0.98 (0.65)	0.98 (0.68)	0.98 (0.71)	0.98 (0.65)	0.98 (0.68)	0.98 (0.71)	1.40 (2.84)	1.42 (2.66)	1.52 (2.44)
64	32.00	29.70	13.60	0.99 (0.67)	0.99 (0.68)	0.99 (0.74)	0.99 (0.67)	0.99 (0.68)	0.99 (0.74)	1.22 (2.53)	1.22 (2.47)	1.21 (2.17)
128	30.90	26.80	13.50	0.99 (0.67)	0.99 (0.69)	1.00 (0.74)	0.99 (0.67)	0.99 (0.69)	0.99 (0.74)	1.19 (2.38)	1.19 (2.28)	1.11 (2.09)
256	30.80	26.00	13.50	0.99 (0.68)	0.99 (0.70)	1.00 (0.74)	0.99 (0.68)	0.99 (0.70)	0.99 (0.74)	1.12 (2.34)	1.12 (2.22)	1.03 (2.04)
normal												
16	37.60	26.40	25.80	0.97 (0.64)	0.97 (0.69)	0.98 (0.69)	0.97 (0.64)	0.97 (0.69)	0.98 (0.69)	1.68 (3.93)	1.64 (3.48)	1.63 (3.46)
32	35.60	30.20	19.00	0.99 (0.64)	0.99 (0.67)	0.99 (0.71)	0.99 (0.64)	0.99 (0.67)	0.99 (0.71)	1.28 (3.12)	1.26 (2.94)	1.29 (2.60)
64	31.50	29.00	15.10	0.98 (0.67)	0.98 (0.68)	0.99 (0.73)	0.98 (0.67)	0.98 (0.68)	0.99 (0.73)	1.32 (2.59)	1.22 (2.61)	1.30 (2.28)
128	31.30	27.10	14.90	0.99 (0.68)	0.99 (0.69)	0.99 (0.74)	0.98 (0.68)	0.99 (0.70)	0.99 (0.74)	1.16 (2.44)	1.16 (2.33)	1.12 (2.12)
256	30.40	27.50	14.60	0.99 (0.68)	0.99 (0.70)	0.99 (0.74)	0.99 (0.68)	0.98 (0.69)	0.98 (0.74)	1.16 (2.33)	1.17 (2.25)	1.22 (2.02)
horseshoe												
16	11.80	14.20	18.80	0.77 (0.46)	0.78 (0.46)	0.80 (0.46)	0.77 (0.48)	0.78 (0.48)	0.80 (0.48)	59.01 (24.57)	62.30 (20.58)	61.01 (29.51)
32	11.90	13.60	20.50	0.77 (0.42)	0.78 (0.42)	0.81 (0.44)	0.77 (0.46)	0.78 (0.45)	0.81 (0.46)	46.55 (19.56)	46.01 (23.46)	49.98 (21.40)
64	12.30	13.60	20.60	0.77 (0.45)	0.78 (0.43)	0.81 (0.45)	0.77 (0.47)	0.78 (0.45)	0.81 (0.48)	41.31 (20.70)	41.55 (21.39)	43.22 (22.99)
128	11.80	13.40	21.00	0.78 (0.42)	0.78 (0.43)	0.81 (0.46)	0.77 (0.45)	0.78 (0.45)	0.81 (0.48)	36.38 (16.88)	36.09 (19.54)	39.55 (17.28)
256	11.90	13.90	21.10	0.77 (0.44)	0.78 (0.44)	0.81 (0.45)	0.77 (0.46)	0.78 (0.46)	0.81 (0.47)	33.13 (12.97)	33.15 (14.78)	35.16 (15.65)

Table 5.13: **Generalization and usability metrics for Gemma-3 with varying MC samples**. The table shows performance for the best-known configurations (Laplace and normal priors with $\sigma = 2.0$) and horseshoe prior for comparison. Values in parentheses represent metrics for non-confident predictions

MC Samples	Confident (%)			Accuracy			F1			PPL		
	95%	90%	50%	95%	90%	50%	95%	90%	50%	95%	90%	50%
Laplace												
16	7.50	6.60	2.40	0.91 (0.50)	0.94 (0.50)	0.98 (0.52)	0.89 (0.46)	0.93 (0.47)	0.97 (0.48)	2.21 (12.19)	1.93 (12.09)	1.09 (11.35)
32	7.30	5.10	1.20	0.89 (0.51)	0.96 (0.52)	1.00 (0.54)	0.87 (0.48)	0.95 (0.48)	1.00 (0.50)	2.95 (4.92)	1.88 (4.99)	1.00 (4.84)
64	5.90	5.10	1.80	0.98 (0.52)	0.99 (0.52)	1.00 (0.54)	0.97 (0.49)	0.98 (0.49)	1.00 (0.50)	1.12 (4.32)	1.08 (4.28)	1.01 (4.09)
128	5.40	4.70	0.30	0.98 (0.51)	0.99 (0.51)	1.00 (0.53)	0.97 (0.47)	0.98 (0.47)	1.00 (0.49)	1.26 (3.92)	1.25 (3.89)	1.01 (3.71)
256	5.40	4.90	0.40	0.99 (0.52)	1.00 (0.52)	1.00 (0.54)	0.99 (0.48)	1.00 (0.48)	1.00 (0.50)	1.08 (3.72)	1.05 (3.71)	1.02 (3.50)
normal												
16	9.80	8.20	3.80	0.86 (0.50)	0.90 (0.51)	0.85 (0.53)	0.82 (0.47)	0.87 (0.47)	0.78 (0.49)	9.18 (11.48)	6.42 (11.80)	19.15 (10.99)
32	7.20	5.20	0.70	0.94 (0.51)	0.96 (0.52)	1.00 (0.54)	0.92 (0.48)	0.95 (0.48)	1.00 (0.50)	2.15 (5.20)	1.56 (5.19)	1.00 (4.93)
64	5.50	4.80	1.00	0.98 (0.52)	0.99 (0.52)	1.00 (0.54)	0.97 (0.48)	0.99 (0.49)	1.00 (0.50)	1.14 (5.19)	1.10 (5.14)	1.01 (4.85)
128	5.60	4.60	0.60	0.99 (0.51)	0.99 (0.52)	1.00 (0.54)	0.99 (0.48)	0.98 (0.48)	1.00 (0.50)	1.09 (3.96)	1.10 (3.90)	1.01 (3.71)
256	5.40	4.90	0.10	0.99 (0.51)	0.99 (0.52)	1.00 (0.54)	0.98 (0.47)	0.98 (0.48)	1.00 (0.50)	1.10 (3.55)	1.10 (3.53)	1.01 (3.34)
horseshoe												
16	86.00	82.90	70.40	0.49 (0.36)	0.50 (0.36)	0.53 (0.36)	0.41 (0.38)	0.41 (0.39)	0.42 (0.37)	3516.42 (29.60)	4203.77 (29.74)	7057.24 (70.22)
32	86.20	83.20	69.10	0.50 (0.37)	0.50 (0.36)	0.53 (0.37)	0.42 (0.40)	0.42 (0.40)	0.42 (0.39)	2860.72 (25.86)	3272.61 (31.26)	6603.47 (53.78)
64	85.40	82.50	68.90	0.50 (0.37)	0.50 (0.37)	0.53 (0.37)	0.41 (0.40)	0.41 (0.40)	0.42 (0.40)	2331.27 (20.72)	2664.65 (24.05)	5007.42 (46.46)
128	85.20	82.30	69.00	0.50 (0.36)	0.51 (0.35)	0.53 (0.35)	0.41 (0.38)	0.42 (0.38)	0.43 (0.37)	2020.85 (17.39)	2316.64 (20.01)	4106.72 (42.96)
256	85.10	82.00	68.40	0.50 (0.37)	0.51 (0.36)	0.53 (0.36)	0.41 (0.39)	0.42 (0.38)	0.43 (0.38)	1737.73 (16.86)	1887.57 (25.76)	3477.05 (43.48)

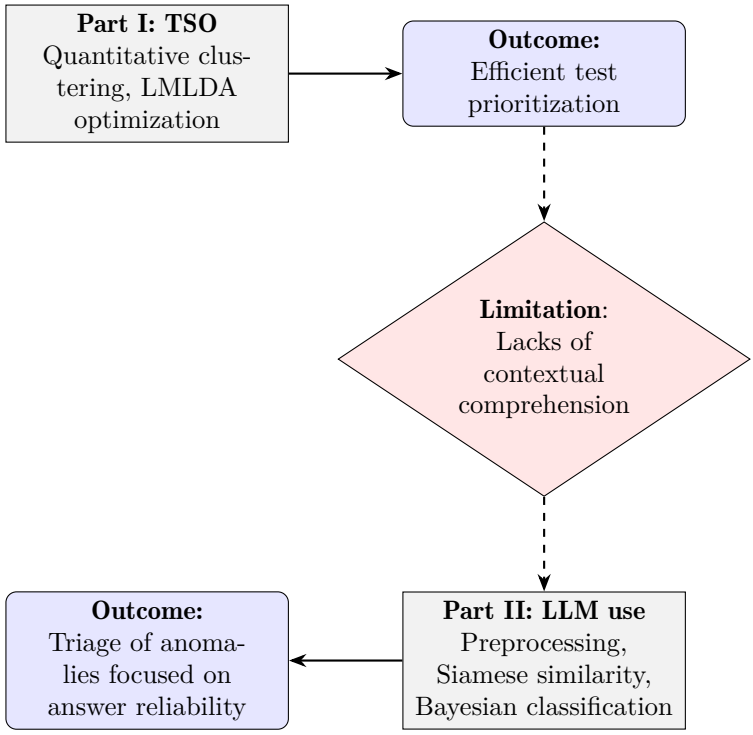
6 Conclusion

This dissertation, the result of an **industrial doctorate** in partnership with **AGH University** and **NOKIA**, confronted the challenges of software quality assurance in critical 4G and 5G telecommunication systems. The research aimed to go beyond theoretical exploration by designing, implementing and validating proprietary ML frameworks directly within an industrial CI/CD pipeline. In the result, the compact overview of the dissertation's research pipeline in Figure 6.1 provides the general idea of the effort and achievements. The primary thesis, that automated anomaly detection and classification could be made both significantly more efficient and significantly more reliable, was validated through successful industrial deployment.

To address the goal of efficiency, this work introduced **LMLDA**, a resource-efficient model for TSO integrated into the pre-deployment testing pipeline. By analyzing latent topics within code changes and test descriptions, LMLDA predicts which tests are most likely to detect anomalies, enabling more focused testing cycles. In its industrial application, the model was introduced into the regression testing pipeline at NOKIA to help teams prioritize their efforts. It achieved **88% bug coverage** and, by restructuring the test sequence to run potentially problematic tests first, it **reduced the average bug discovery time by 8 hours daily**, allowing faster fixing.

To address the need for reliability, a sophisticated **Bayesian LLM framework** was developed to identify detected software anomalies. The first stage of the

Figure 6.1: Compact overview of the dissertation's research pipeline



two-step process uses LoRA for efficient and domain-specific feature extraction, while the second stage uses a Bayesian classification head to provide reliable uncertainty estimates, effectively managing epistemic uncertainty. The key findings of this research established that the foundational architecture of a LLM is the primary limiter of performance and that priors with lighter tails (e.g. normal, laplace) are more effective for dense embeddings. When implemented to optimize the handling of incoming software anomaly reports, this framework successfully created a reliable HITL system. The best model configurations could **automatically address nearly 40% of tickets with 98% precision**, freeing engineers from repetitive work and deferring ambiguous cases to human experts.

The successful integration of these frameworks into the live software development processes for the 4G and 5G base stations at NOKIA confirms the measur-

able success and real-world value of this work. Beyond the technical performance, these tools helped address organizational challenges, such as resistance to new technologies in established processes. By providing reliable and interpretable results, particularly the Bayesian framework's ability to communicate uncertainty and avoid silent failures, the systems improved engineer trust and demonstrated clear value. The results, summarized in Table 6.1, confirm the practical viability and industrial impact of this research.

Table 6.1: Assessment of the implemented ML systems in an industrial context

Characteristic	Assessment	Outcome
Effectiveness	Achieved specific goals of automating triage and optimizing testing.	~40% of tickets automated with 98% precision. Excellent bug coverage of 88% and faster detection from LMLDA (8 hours in daily routine).
Satisfaction	Positive reception from engineers due to system reliability and usefulness.	Enhanced trust in the tool due to its ability to communicate uncertainty and avoid silent failures.
Freedom from risk	Mitigated the risk of critical bugs being misrouted or delayed.	Uncertain tickets are automatically flagged for human review, minimizing incorrect assignments.
Context coverage	Frameworks were successfully applied to the complex, real-world data of 4G and 5G software development.	Models trained and validated on thousands of internal bug reports and test cases.

6.1 Consolidated research questions and findings

The core contributions of this dissertation can be distilled into the answers to a series of targeted research questions that guided the work from a TSO to a reliable LLM-based triage.

- **RQ 1:** To what extent can a purely quantitative delta-centric clustering approach effectively select failing tests and what are its primary limitations?

  Answer: This approach identifies 70-80% common faults with a low TPR, but fails in rarer defects due to its primary limitation: a complete lack of semantic context with respect to the nature of the changes in the code.

- **RQ 2:** Can a generative classifier, such as LMLDA, that models the semantic relationship between textual content significantly improve test prioritization and accelerate defect discovery?

  Answer: Yes. The LMLDA model achieved a high TPR of 87% and, when used for prioritization, created a measurable left-shift in defect discovery, finding critical bugs up to a full working day earlier (approx. 8 hours).

- **RQ 3:** How effectively can a fine-tuned Siamese network architecture identify semantically duplicate bug reports within a specialized industrial domain?

  Answer: It is highly effective. After fine-tuning, the Siamese network correctly identified over 81% similar pair of tickets, demonstrating its ability to mitigate the problem of duplicate anomalies between groups of developers.

- **RQ 4:** Is linguistic adaptation (measured by perplexity reduction) a sufficient condition to achieve high performance in a downstream classification task when fine-tuning an LLM trough prompt-engineering?

  Answer: No. While using a prompt=engineering, reducing perplexity by more than 80% only increased the classification accuracy by 10% (from 20% to 30%), demonstrating that linguistic fluency does not guarantee task competence and highlighting the challenge of epistemic uncertainty.

- **RQ 5:** What is the impact of the prior distribution scale (σ) on model performance in a Bayesian classification head?

  Answer: For pre-trained LLM feature extractors, a relaxed prior (for example, $\sigma = 2.0$) is critical. Narrow priors ($\sigma = 0.01$) cause severe under-fitting and poor calibration, indicating that over-regularization is a greater risk than under-regularization in this context.

- **RQ 6:** Which Bayesian prior (normal, Laplace, horseshoe) is most effective for this task?

  Answer: The Laplace prior consistently offers a slight advantage, providing the best calibration (lowest ECE) by effectively pruning redundant features

without being overly aggressive. The complex horseshoe prior consistently underperformed and yielded overconfident and unreliable models.

- **RQ 7:** How do the underlying LLM architectural choices interact with the Bayesian framework?

  Answer: The attention mechanism is the most critical factor. GQA architectures (Llama-3.2, Qwen3) consistently outperformed the Multi-Head (Deepseek-Coder) and Multi-Query (Gemma-3) variants.

- **RQ 8:** Can this proposed framework form the basis of a reliable, production-ready system for automated triage?

  Answer: Yes. The combination of the Qwen3 model with a Laplace prior ($\sigma = 2.0$) creates a robust HITL system. It can automate 36.6% of incoming tickets with 98% F1 accuracy while flagging the remaining uncertain cases for human review, thus satisfying the industrial need for both efficiency and reliability.

6.2 Future Work

This work can be extended in several practical directions to improve its performance and applicability in an operational engineering environment. The following areas are identified for future investigation.

- **Reducing inference latency for uncertainty quantification**: The current inference process for uncertainty quantification is computationally expensive, which poses a challenge for real-time applications. Future work should investigate more efficient approximate Bayesian methods, such as normalizing flows or alternative variational inference schemes, to lower the latency and computational resource requirements. The objective is to reduce the cost of inference without a substantial loss in the quality of the uncertainty estimates.

- **Explanations for uncertainty**: The current system can report *that* it is uncertain, but not *why*. A key next step is to integrate explainability techniques that attribute model uncertainty to specific features in input data.

Highlighting the words, phrases, or data points that contribute most to uncertainty would provide engineers with direct, actionable feedback to diagnose model behavior or correct data quality issues.

- **Handling multi-modal data**: Bug reports often contain more than unstructured text. The framework should be extended to a multi-modal system capable of processing associated artifacts like system logs, stack traces, and performance metrics. Furthermore, to address the data drift inherent in a continuous CI/CD pipeline, methods for online learning or efficient model update are needed to maintain performance over time and reduce the operational overhead of periodic full-model retraining.

Bibliography

[1] 3GPP. Vocabulary for 3GPP Specifications. Technical Specification (TS) 21.905, 3rd Generation Partnership Project (3GPP), 2021. Version 17.1.0.

[2] R. Abdalkareem, S. Mujahid, and E. Shihab. A Machine Learning Approach to Improve the Detection of CI Skip Commits. *IEEE Transactions on Software Engineering*, 47:2740–2754, 2021.

[3] M. Alenezi and S. Banitaan. Bug reports prioritization: Which features and classifier to use. In *Proc. 12th Int. Conf. Mach. Learn. Appl.*, volume 2, pages 112–116, 2013.

[4] K. Alreshedy, D. Dharmaretnam, D. M. German, V. Srinivasan, and T. A. Gulliver. SCC: Automatic classification of code snippets. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 203–208, 2018.

[5] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2017.

[6] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 361–370. Association for Computing Machinery, 2006.

[7] M. J. Arafeen and H. Do. Test case prioritization using requirements-based clustering. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 312–321, 2013.

[8] K. Asadi and M. Littman. An Alternative Softmax Operator for Reinforcement Learning. In *Proceedings of The 34th International Conference on Machine Learning*, volume 70, pages 243–252. JMLR.org, 8 2017.

[9] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *The Semantic Web*, pages 722–735, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[10] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[11] Y. Bai, S. Kadavath, S. Kundu, et al. Constitutional AI: Harmlessness from AI Feedback. *arXiv*, 2022.

[12] K. Beck, M. Beedle, A. van Binsbergen, et al. The Agile Manifesto. `http://agilemanifesto.org/`, 2001.

[13] I. Beltagy, K. Lo, and A. Cohan. SciBERT: A pretrained language model for scientific text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3615–3620, Hong Kong, China, 11 2019. Association for Computational Linguistics.

[14] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1–12, 2020.

[15] S. Bird, R. Dale, B. Dorr, et al. The ACL Anthology reference corpus: A reference dataset for bibliographic research in computational linguistics. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, pages 1755–1759, Marrakech, Morocco, 05 2008.

[16] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[17] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight uncertainty in neural networks. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1613–1622, Lille, France, July 2015. PMLR.

[18] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–147, 2017.

[19] A.-L. Bornea, F. Ayed, A. De Domenico, N. Piovesan, and A. Maatouk. Telco-rag: Navigating the challenges of retrieval augmented language models for telecommunications. In *GLOBECOM 2024 - 2024 IEEE Global Communications Conference*, pages 2359–2364, 2024.

[20] T. Brown, B. Mann, N. Ryder, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[21] B. Busjaeger and T. Xie. Learning for test prioritization: an industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 975–980, New York, NY, USA, 2016. Association for Computing Machinery.

[22] R. Carlson, H. Do, and A. Denton. A clustering approach to improving test case prioritization: An industrial case study. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 382–391, 2011.

[23] Y. C. Cavalcanti, M. S. Neto, P. Anselmo, et al. Challenges and opportunities for software change request repositories: A systematic mapping study. *Journal of Software: Evolution and Process*, 26(7):620–653, July 2014.

[24] J. Chen, Y. Gu, S. Cai, H. Chen, and J. Chen. Ks-tcp: An efficient test case prioritization approach based on k-medoids and similarity. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 105–110, 2021.

[25] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang. Optimizing test prioritization via test distribution analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 656–667, New York, NY, USA, 2018. Association for Computing Machinery.

[26] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng. Using semi-supervised clustering to improve regression test selection techniques. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10, 2011.

[27] T. Chen, S. Thomas, H. Hemmati, M. Nagappan, and A. Hassan. An empirical study on the effect of testing on code quality using topic models: A case study on software development systems. *IEEE Transactions on Reliability*, 66:806–824, 2017.

[28] N. Chetouane, F. Wotawa, H. Felbinger, and M. Nica. On using k-means clustering for test suite reduction. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 380–385, 2020.

[29] Choudhary. Neural network based bug priority prediction model using text classification techniques. *Int. J. Adv. Res. Comput. Sci.*, 8(5):1315–1319, 2017.

[30] A. Cohan, W. Ammar, M. van Zuylen, and F. Cady. Structural scaffolds for citation intent classification in scientific publications. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3586–3596, Minneapolis, Minnesota, 06 2019. Association for Computational Linguistics.

[31] C. Coviello, S. Romano, G. Scanniello, A. Marchetto, G. Antoniol, and A. Corazza. Clustering support for inadequate test suite reduction. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–105, 2018.

[32] D. Cubranic and G. Murphy. Automatic bug triage using text categorization. In *SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 01 2004.

[33] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais. Effective test generation using pre-trained large language models and mutation testing, 2024.

[34] D. L. Davies and D. W. Bouldin. A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (2):224–227, 1979.

[35] K. Dejaeger, T. Verbraken, and B. Baesens. Toward Comprehensible Software Fault Prediction Models Using Bayesian Network Classifiers. *IEEE Transactions on Software Engineering*, 39:237–257, 2013.

[36] G. Di Lucca, M. Di Penta, and S. Gradara. An approach to classify software maintenance requests. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 93–102, 2002.

[37] N. Ding, Y. Qin, G. Yang, et al. Parameter-efficient fine-tuning of large-scale pre-trained language models, Mar 2023.

[38] S. Elbaum, G. Malishevsky, G. Rothermel, and R. H. Untch. Evaluating the effectiveness of regression test prioritization techniques. In *Proceedings of the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 255–266, 2000.

[39] Z. Feng, D. Guo, D. Tang, et al. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, Nov. 2020. Association for Computational Linguistics.

[40] R. A. Fisher. The logic of inductive inference. *Journal of the Royal Statistical Society*, 98(1):39–54, 12 2018.

[41] M. Fomicheva, S. Sun, L. Yankovskaya, F. Blain, F. Guzmán, M. Fishel, N. Aletras, V. Chaudhary, and L. Specia. Unsupervised quality estimation for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:539–555, 2020.

[42] P. Gage. A new algorithm for data compression. *C Users J.*, 12(2):23–38, Feb. 1994.

[43] Y. Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, 2016.

[44] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. 48:1050–1059, 20–22 Jun 2016.

[45] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1050–1059. JMLR.org, 2016.

[46] T. Glushkova, C. Zerva, R. Rei, and A. F. T. Martins. Uncertainty-aware machine translation evaluation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3920–3938, Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics.

[47] L. Gong, H. Zhang, J. Zhang, M. Wei, and Z. Huang. A Comprehensive Investigation of the Impact of Class Overlap on Software Defect Prediction. *IEEE Transactions on Software Engineering*, 49:2440–2458, 2023.

[48] G. Grano, T. V. Titov, S. Panichella, and H. C. Gall. How high will it be? using machine learning models to predict branch coverage in automated testing. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 19–24, 2018.

[49] W. Grathwohl, K. Wang, J. Jacobsen, D. Duvenaud, M. Norouzi, and K. Swersky. Your Classifier is Secretly an Energy Based Model and You Should Treat it Like One. *ArXiv*, abs/1912.03263, 2019.

[50] A. Grattafiori, A. Dubey, A. Jauhri, et al. The llama 3 herd of models, 2024.

[51] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, Vancouver, BC, Canada, 2013. IEEE.

[52] D. Guo, Q. Zhu, D. Yang, et al. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

[53] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 06(02):107–116, 1998.

[54] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[55] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models. In *ICLR*. OpenReview.net, 2022.

[56] Y. Huang, H. Du, X. Zhang, D. Niyato, J. Kang, Z. Xiong, S. Wang, and T. Huang. Large language models for networking: Applications, enabling techniques, and challenges. *IEEE Network*, 39(1):235–242, 2025.

[57] E. Hüllermeier and W. Waegeman. Aleatoric and epistemic uncertainty in machine learning: an introduction to concepts and methods. *Machine Learning*, 110:457–506, 2021.

[58] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker. Perplexity of the language model. *IBM Thomas J. Watson Research Center, Yorktown Heights, NY*, 1977.

[59] N. Jiang, K. Liu, T. Lutellier, and L. Tan. Impact of Code Language Models on Automated Program Repair. In *Proceedings of The 45th International Conference on Software Engineering*, pages 1430–1442, 2023.

[60] L. Jonsson, D. Broman, M. Magnusson, K. Sandahl, M. Villani, and S. Eldh. Automatic localization of bugs to faulty components in large scale software systems using bayesian classification. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 423–430, 2016.

[61] L. Jonsson, D. Broman, K. Sandahl, and S. Eldh. Towards automated anomaly report assignment in large complex systems using stacked generalization. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 437–446, 2012.

[62] W. Jun and F. Meng. Software testing based on cloud computing. In *2011 International Conference on Internet Computing and Information Services*, pages 176–178, 2011.

[63] R. Kallis, A. Di Sorbo, G. Canfora, and S. Panichella. Ticket tagger: Machine learning driven issue classification. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 406–409, 2019.

[64] A. Karapantelakis, M. Thakur, A. Nikou, and V. others. Using Large Language Models to Understand Telecom Standards. In *2024 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, pages 440–446, 2024.

[65] A. Kendall and Y. Gal. What uncertainties do we need in bayesian deep learning for computer vision? In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[66] T. Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[67] T. Kudo and J. Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In

Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pages 66–71, Brussels, Belgium, Nov. 2018. Association for Computational Linguistics.

[68] L. Kuhn, Y. Gal, and S. Farquhar. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation, 2023.

[69] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer. System-level test case prioritization using machine learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 361–368, 2016.

[70] T. Le, F. Thung, and D. Lo. Will this localization tool be effective for this bug? mitigating the impact of unreliability of information retrieval based bug localization tools. *Empirical Software Engineering*, 22:2237–2279, 2017.

[71] A. Lenz, A. Pozo, and S. Vergilio. Linking software testing results with a machine learning approach. *Engineering Applications of Artificial Intelligence*, 26:1631–1640, 5 2013.

[72] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[73] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[74] G. Li, L. Yang, C.-G. Lee, X. Wang, and M. Rong. A bayesian deep learning rul framework integrating epistemic and aleatoric uncertainties. *IEEE Transactions on Industrial Electronics*, 68(9):8829–8841, 2021.

[75] L. Li, Z. Li, W. Zhang, J. Zhou, P. Wang, J. Wu, G. He, X. Zeng, Y. Deng, and T. Xie. Clustering test steps in natural language toward automating

test automation. In *Proceedings of The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on The Foundations of Software Engineering*, pages 1285–1295, 2020.

[76] F. Liu, J. Zhang, and E.-Z. Zhu. Test-suite reduction based on k-medoids clustering algorithm. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 186–192, 2017.

[77] S. Liu, H. Tao, and S. Feng. Text classification research based on bert model and bayesian network. In *2019 Chinese Automation Congress (CAC)*, pages 5842–5846, 2019.

[78] T. Louis. Finding the Observed Information Matrix when Using the EM Algorithm. *Journal of The Royal Statistical Society. Series B (Methodological)*, 44:226–233, 1982.

[79] J. Lousada and M. Ribeiro. Neural network embeddings for test case prioritization, 2020.

[80] T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal, Sept. 2015. Association for Computational Linguistics.

[81] M. Mahdieh, S.-H. Mirian-Hosseinabadi, K. Etemadi, A. Nosrati, and S. Jalali. Incorporating fault-proneness estimations into coverage-based test case prioritization methods. *Information and Software Technology*, 121:106269, 2020.

[82] M. Mahdieh, S.-H. Mirian-Hosseinabadi, and M. Mahdieh. Test case prioritization using test case diversification and fault-proneness estimations. *Automated Software Engineering*, 29(2):50, 2022.

[83] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[84] D. Marijan, A. Gotlieb, and A. Sapkota. Neural network classification for improving continuous regression testing. In *Proc. IEEE Int. Conf. Artif. Intell. Test. (AITest)*, pages 123–124, Aug 2020.

[85] L. McInnes, J. Healy, and J. Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2020.

[86] N. Medhat, S. M. Moussa, N. L. Badr, and M. F. Tolba. A framework for continuous regression and integration testing in iot systems based on deep learning and search-based techniques. *IEEE Access*, 8:215716–215726, 2020.

[87] C. Meister and R. Cotterell. Language model evaluation beyond perplexity. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5328–5339. Association for Computational Linguistics, 2021.

[88] A. Miaschi, D. Brunato, F. Dell'Orletta, and G. Venturi. What makes my model perplexed? a linguistic investigation on neural language models perplexity. In *Proceedings of Deep Learning Inside Out (DeeLIO): The 2nd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 40–47, Online, June 2021. Association for Computational Linguistics.

[89] G. Myers, C. Sandler, and T. Badgett. *The art of software testing.* John Wiley & Sons, 2012.

[90] A. Occhipinti, L. Rogers, and C. Angione. A pipeline and comparative study of 12 machine learning models for text classification. *Expert Systems with Applications*, 201:117193, 2022.

[91] I. Osband, Z. Wen, S. M. Asghari, et al. Epistemic neural networks. In *Advances in Neural Information Processing Systems*, volume 36, pages 2795–2823. Curran Associates, Inc., 2023.

[92] F. Palma, T. Abdou, A. Bener, J. Maidens, and S. Liu. An improvement to test case failure prediction in the context of test case prioritization. In

Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, pages 80–89, 2018.

[93] L. Pan, Q. Cai, Q. Meng, W. Chen, and L. Huang. Reinforcement learning with dynamic Boltzmann softmax updates, 2021.

[94] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 199–209. Association for Computing Machinery, 2011.

[95] H. Pei, B. Yin, M. Xie, and K. Cai. Dynamic random testing with test case clustering and distance-based parameter adjustment. *Information and Software Technology*, 131:106470, 2021.

[96] Y. Peng, S. Yan, and Z. Lu. Transfer Learning in Biomedical Natural Language Processing: An Evaluation of BERT and ELMo on Ten Benchmarking Datasets, 2019.

[97] A. V. Phan, M. Le Nguyen, and L. T. Bui. Convolutional neural networks over control flow graphs for software defect prediction, 2017.

[98] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[99] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training, 2018.

[100] L. E. Raileanu and K. Stoffel. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, 2004.

[101] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, 11 2016. Association for Computational Linguistics.

[102] S. Reddy, D. Chen, and C. D. Manning. CoQA: A Conversational Question Answering Challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266, 05 2019.

[103] P. J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[104] I. Saidani, A. Ouni, and M. W. Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Softw. Eng.*, 29:21, Jan 2022.

[105] H. P. Samoaa, A. Longa, M. Mohamad, M. H. Chehreghani, and P. Leitner. Tep-gnn: Accurate execution time prediction of functional tests using graph neural networks. In *Proc. Int. Conf. Product-Focused Softw. Process Improvement*, pages 464–479, 2022.

[106] A. Sarkar, P. C. Rigby, and B. Bartalos. Improving bug triaging with high confidence predictions at ericsson. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 81–91, 2019.

[107] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering*, 50:85–105, 2024.

[108] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 1715–1725, 2016.

[109] M. Shahin, M. Ali Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[110] A. Sharif, D. Marijan, and M. Liaaen. Deeporder: Deep learning for test case prioritization in continuous integration testing. In *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, pages 525–534, Sep 2021.

[111] K. Shimari, M. Tanaka, T. Ishio, M. Matsushita, K. Inoue, and S. Takanezawa. Selecting test cases based on similarity of runtime information: A case study of an industrial simulator. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 564–567, 2022.

[112] C. Silva, M. Galster, and F. Gilson. Topic modeling in software engineering research. *Empirical Software Engineering*, 26:120, Sep 2021.

[113] M. M. Suarez-Alvarez, D.-T. Pham, M. Y. Prostov, and Y. I. Prostov. Statistical approach to normalization of feature vectors and clustering of mixed datasets. *Proceedings of the Royal Society A*, 468:2630–2651, 2012.

[114] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[115] S. Tahvili, L. Hatvani, M. Felderer, W. Afzal, and M. Bohlin. Automated functional dependency detection between test cases using doc2vec and clustering. In *Proc. IEEE Int. Conf. Artif. Intell. Test. (AITest)*, pages 19–26, Apr 2019.

[116] H. Takeuchi and I. Nonaka. The new new product development game. *Harvard Business Review*, 64(1):137–146, 1986.

[117] A. Talman, H. Celikkanat, S. Virpioja, M. Heinonen, and J. Tiedemann. Uncertainty-aware natural language inference with stochastic weight averaging. In *Proceedings of the 24th Nordic Conference on Computational Linguistics (NoDaLiDa)*, pages 358–365, Tórshavn, Faroe Islands, May 2023. University of Tartu Library.

[118] G. Team, A. Kamath, J. Ferret, et al. Gemma 3 technical report, 2025.

[119] The Copenhagen Post. Massive outage disrupts tdc mobile services, emergency calls affected. `https://cphpost.dk/2024-11-28/news/round-up/massive-outage-disrupts-tdc-mobile-services-emergency-calls-affected/`, November 2024. Accessed on November 28, 2024.

[120] S. Thomas, H. Hemmati, A. Hassan, and D. Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, 19:182–212, 2014.

[121] F. Tu, J. Zhu, Q. Zheng, and M. Zhou. Be careful of when: an empirical study on time-related misuse of issue tracking data. In *Proceedings of The 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on The Foundations of Software Engineering*, pages 307–318, 2018.

[122] D. Ulmer, J. Frellsen, and C. Hardmeier. Exploring predictive uncertainty and calibration in NLP: A study on the impact of method & data scarcity, Dec. 2022.

[123] Q. Umer, H. Liu, and I. Illahi. CNN-based automatic prioritization of bug reports. *IEEE Transactions on Reliability*, 69(4):1341–1354, 2020.

[124] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[125] A. Vazhentsev, G. Kuzmin, A. Shelmanov, et al. Uncertainty estimation of transformer predictions for misclassification detection. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8237–8252, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[126] M. Viggiato, D. Paas, C. Buzon, and C.-P. Bezemer. Using natural language processing techniques to improve manual test case descriptions. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 311–320, 2022.

[127] M. Viggiato, D. Paas, C. Buzon, and C.-P. Bezemer. Identifying similar test cases that are specified in natural language. *IEEE Transactions on Software Engineering*, 49(3):1027–1043, 2023.

[128] B. Wang. Disconnected recurrent neural networks for text categorization. In *Proc. 56th Annu. Meeting Assoc. Comput. Linguistics (Volume 1: Long Papers)*, pages 2311–2320, Jul 2018.

[129] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50:911–936, 2024.

[130] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22. Curran Associates Inc., 2022.

[131] C. Whittaker, Arbon. *How Google Tests Software*. Addison-Wesley Professional, 2012.

[132] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, N. Jaitly, I. Sutskever, G. S. Corrado, J. Dean, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. In *arXiv preprint arXiv:1609.08144*, 2016.

[133] C. Xia, Y. Wei, and L. Zhang. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of The 45th International Conference on Software Engineering*, pages 1482–1494, 2023.

[134] X. Xia, D. Lo, Y. Ding, J. Al-Kofahi, T. Nguyen, and X. Wang. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering*, 43:272–297, 2017.

[135] L. Xiao, H. Miao, T. Shi, and Y. Hong. LSTM-based deep learning for spatial–temporal software testing. *Distributed and Parallel Databases*, 38(3):687–712, Sep 2020.

[136] Y. Xiao, P. P. Liang, U. Bhatt, W. Neiswanger, R. Salakhutdinov, and L.-P. Morency. Uncertainty quantification with pre-trained language models:

A large-scale empirical analysis. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 7273–7284, Abu Dhabi, United Arab Emirates, Dec. 2022. Association for Computational Linguistics.

[137] A. Yang, A. Li, B. Yang, et al. Qwen3 technical report, 2025.

[138] A. X. Yang, M. Robeyns, X. Wang, and L. Aitchison. Bayesian low-rank adaptation for large language models. In *International Conference on Learning Representations*, 2024.

[139] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 201–212, 2009.

[140] X. Yu, K. Jia, W. Hu, J. Tian, and J. Xiang. Black-box test case prioritization using log analysis and test case diversity. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 186–191, 2023.

[141] S. Zarębski, M. Jokiej, M. Szczukiewicz, K. Rusek, and P. Chołda. An unsupervised machine learning approach for regression testing queue optimisation in 5G base stations. *Przegląd Telekomunikacyjny, Wiadomości Telekomunikacyjne*, 96(4):208–211, 2023.

[142] S. Zarębski, A. Kuzmich, S. Sitko, K. Rusek, and P. Chołda. Siamese neural networks on the trail of similarity in bugs in 5g mobile network base stations. *Electronics*, 11(22), 2022.

[143] C. Zhang, Z. Chen, Z. Zhao, S. Yan, J. Zhang, and B. Xu. An improved regression test selection technique by clustering execution profiles. In *2010 10th International Conference on Quality Software*, pages 171–179, 2010.

[144] H. Zhang, Y.-F. Zhang, Y. Yu, D. Madeka, D. Foster, E. Xing, H. Lakkaraju, and S. Kakade. A study on the calibration of in-context learning. In *Proceedings of the 2024 Conference of the North American*

Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 6118–6136, Mexico City, Mexico, June 2024. Association for Computational Linguistics.

[145] X. Zhao, Z. Wang, X. Fan, and Z. Wang. A clustering-bayesian network based approach for test case prioritization. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 542–547, 2015.