Learning-based classification of software logs generated by a test automation framework

Juri Voloskin

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 28.02.2022

Supervisor

Professor Arno Solin

Advisors

Dr Andrea Pilzer

MSc Mikko Kujala

Aalto University
School of Electrical Engineering

Aalto University
School of Electrical Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Juri Voloskin

Title Learning-based classification of software logs generated by a test automation framework

Degree programme Automation and electrical engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC3025

Supervisor Professor Arno Solin

Advisors Dr Andrea Pilzer, MSc Mikko Kujala

Date 28.02.2022 **Number of pages** 89+4 **Language** English

Abstract

Managing large software development systems has become increasingly challenging, as large volumes of raw data generated by the production telemetry are intractable for manual processing. The client of this thesis seeks an effective scalable approach to tackle this issue by automatically classifying the software logs generated in case of integration test failures during software production.

This thesis has developed two machine learning candidate solutions to demonstrate the feasibility of a learning-based approach for log classification. The first solution represents a canonical natural language processing pipeline, which performs step-by-step transformation of the input data using text preprocessing and numerical representation methods as well as permits using any traditional machine learning model for classification. The second solution employs the transfer learning approach and a deep neural language model from the family of bidirectional transformers, which incorporates an encoder for contextual text representation that is fine-tuned on a domain-specific corpus to improve classification performance.

Both solutions achieved high accuracy scores, thus confirming the feasibility of a learning-based approach for software log classification. Experiments showed that contextual text representations using no text preprocessing contributed more to classification accuracy than other representation schemes attempted in this work. A transformer neural language model pre-trained on the general natural language domain successfully adapted to the domain of software logs with minimal preprocessing effort. At the same time, the experimental results indicated that careful vocabulary management and methodical log preprocessing could enhance similarity between the domains and thus further improve the classification accuracy of the transfer learning solution.

Keywords Artificial Intelligence, Machine Learning, Deep Learning, Natural Language Processing, Continuous Integration, Test Automation

Preface

I wish to express my sincere gratitude to my thesis supervisor Professor Arno Solin for guiding me through the thesis process and providing invaluable advice concerning this thesis structure and content. I am also very grateful to Dr Andrea Pilzer for his constructive feedback on the design choices made during the solution development and to MSc Mikko Kujala for providing insight into the specifics of the client application.

I am deeply grateful to my academic tutor Professor Themistoklis Charalambous for his continuous support throughout my studies at Aalto as well as for encouraging me to pursue this thesis topic and to Dr Annika Salama for her guidance throughout the formal requirements of the degree programme.

Finally, I would like to offer my special thanks to lecturer Kenneth Pennington for his constructive feedback on the writing of this thesis, for his course LC-1320 Thesis Writing for Engineers, and for the material he created to support English language studies at Aalto University.

Otaniemi, 28.02.2022

Juri Voloskin

Contents

Abstract **3**

Preface **4**

Contents **5**

Abbreviations **6**

1 Introduction **7**
1.1 Problem Statement 7

2 Background **10**
2.1 Artificial Intelligence (AI) 10
2.2 Rule-based AI 11
2.3 Machine Learning (ML) 12
2.4 Natural Language Processing (NLP) 18

3 Related Work **21**
3.1 Software Log Analysis and Classification 21
3.2 Transformers in Transfer Learning 23

4 Methods **28**
4.1 Application and Data Acquisition 28
4.2 Baseline: Machine Learning Pipeline 30
4.3 Transfer Learning Solution 51

5 Results **57**
5.1 Baseline Solution Evaluation 57
5.2 Transfer Learning Solution Evaluation 70

6 Discussion **75**
6.1 Comparison of the Solution Candidates 75
6.2 Assessment of the Classification Results 77

7 Conclusion **82**

References **84**

A Appendix 1: Confusion Matrices **90**

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
CI	Continuous Integration
DuT	Device under Test
ERM	Empirical Risk Minimization
GPT	Generative Pre-training Transformers
IT	Information Technology
IQR	Interquartile Range
KNN	K-Nearest Neighbours
LDA	Latent Dirichlet Allocation
LM	Language Model
LSI	Latest Semantic Indexing
LSTM	Long Short-Term Memory (ANN)
MCC	Matthews Correlation Coefficient
ML	Machine Learning
MLE	Maximum Likelihood Estimate
MLM	Masked Language Modelling
NLM	Neural Language Model
NLP	Natural Language Processing
PAC	Probably Approximately Correct (learning)
PCA	Principal Component Analysis
RNN	Recurrent Neural Network
SLM	Statistical Language Model
SVD	Singular Value Decomposition
SVM	Support Vector Machine
TL	Transfer Learning

1 Introduction

Modern software development systems have become increasingly sophisticated. With the widespread adoption of *agile* practices, almost every step in software production has become automated to eliminate the need for manual labour and to streamline the integration of software changes continuously submitted by many collaborators [1]. This approach to software production is known as *continuous integration* (CI), and the instances of such production systems are referred to as *CI pipelines*.

CI pipelines are delicate systems that should be continuously monitored and adjusted to meet the requirements of an ever-evolving software development environment because their uninterrupted work is one of the main factors determining production throughput [2]. To facilitate status telemetry and effective fault tracing, the pipelines keep a record of their activity using *logs* of graphical and textual information. If a break occurs in the workflow, this data becomes instrumental for troubleshooting, as it provides clues for a probable cause of the issue.

Professional software development practice has established a standard for event logging to permeate throughout the software production system to ensure oversight of every system component [2]. On the one hand, such a detailed logging scheme permits localising problems with extreme accuracy. On the other hand, automatically generated logs tend to grow in size, making it challenging and time-consuming to find relevant information.

In the event of pipeline failure, it is a common practice for human operators to manually browse over log data, searching for hints to help them localise the source of a problem and take actions to resolve it. This approach may be acceptable at the early stages of a software production system life cycle when the frequency of failures is relatively low. However, as the software grows more complex and the production system increases in scale, the likelihood of failure increases proportionally [1], thereby making pipeline interruptions more frequent. Consequently, the sheer volume of generated log data makes manual inspection intractable. Therefore, an organisation that has no means of automating the log analysis and failure tracing process is poised to suffer a decline in throughput and prolonged delivery cycles.

1.1 Problem Statement

The client for this thesis, a technology organisation, wishes to improve the problem resolution management in its software development environment by automating the analysis and classification of failed test reports. These reports mainly consist of software logs. Hence, the primary challenge is to find an effective approach to utilise the text information from these logs for the classification task.

The log classification task has a high potential for automation because log contents tend to follow an implicit structure and are often repetitive [3]. The client has already created a solution that uses heuristic rules to extract keywords and identify sentence patterns that can be automatically mapped to a predefined category. However, there are two problems with the current approach. Firstly, it is challenging to design rules that are general enough to cover a great variability of log content.

Secondly, a continuously evolving production system causes the log content to change over time, thus aggravating the first problem. Together, these issues result in a continuous burden of creating more (often conflicting) rules, which eventually renders the rule-based approach intractable [4].

In recent years, multiple publications have addressed the software log classification problem. Several of these studies [5]–[7] have highlighted the learning-based approach as effective in overcoming the issues of rule-based solutions. Indeed, machine learning methods have the potential to eliminate the need for handcrafted rules by automatically extracting statistically significant associations from training examples. Furthermore, the diversity of learned associations enables such a classifier to generate predictions for previously unseen data samples that moderately deviate from the original training set. Finally, as more examples accumulate, the quality of prediction tends to increase [8], thus facilitating *automatic continuous self-improvement* of the learning-based solution.

Despite their merits, conventional learning-based methods often struggle to meet performance requirements if the input data lacks structure and contains excessive noise. This problem is so pernicious in machine learning projects that data cleaning and searching for better forms of representation account for a considerable share of the solution design effort [9]. One possible approach to overcome this problem would be to employ a neural network model with sufficient expressive power to automatically learn an effective data representation and mitigate the noise problem by applying a suitable regularisation technique. Such a neural network solution would simplify the development of a learning-based classifier by alleviating the need for tedious data preprocessing and feature engineering.

Among the neural networks capable of working with sequential text input, neural language models have become particularly effective in a variety of natural language processing tasks [10], including text classification. Notably, the feasibility of applying neural language models to classify unstructured log data originated from a software development environment still remains largely unexplored in the literature.

Therefore, the aim of this thesis is to develop a software solution implementing a learning-based log classifier that would achieve performance comparable to that of the currently used rule-based solution while requiring the same or less supervision effort and obviating the need for handcrafted heuristic rules. Furthermore, this thesis aims to assess the capacity of a pre-trained and fine-tuned neural language model to capture the semantic structure of software logs and produce contextually accurate encoding of unstructured log content, thus aiding classification accuracy.

To accomplish these goals, this thesis will develop the baseline solution implementing a conventional machine learning pipeline that accepts raw labelled data as input, extracts relevant features for learning, converts these features into a suitable mathematical representation, and outputs a trained classifier object for deployment in the field. As an alternative solution, the thesis will adapt a pre-trained neural language model to the domain of software logs and fine-tune it for the classification task, thus decreasing the data preprocessing effort compared to the baseline solution. Finally, the thesis will compare the candidate solutions using standard classifier evaluation metrics.

The remainder of this thesis is organised as follows. Section 2 reviews the concepts essential for this study, such as rule-based approach, machine learning methods, and natural language processing. Section 3 surveys the earlier work concerned with the log classification problem and outlines the recent advancements in neural language models and the related learning strategies. Section 4 describes the solution methods and their step-by-step implementation. Section 5 presents and formally evaluates the experimental results. Section 6 interprets the results and discusses the main findings. Finally, Section 7 concludes the thesis by summarising the solution design and methods, evaluating the proposed solution and giving recommendations for future work.

2 Background

Section 1 has presented the learning-based approach as a potential solution for overcoming the lack of scalability in the rule-based method. Because these approaches, as well as the related study of natural language processing, attempt to mimic the human reasoning process, they are subsumed under the more general field of study, known as artificial intelligence. This chapter takes a deeper look into these methods. Specifically, Section 2.1 introduces the concept of artificial intelligence, Section 2.2 explains the notion of rule-based artificial intelligence, Section 2.3 presents the essential elements of machine learning theory, including deep learning, and Section 2.4 introduces the relevant concepts from the field of natural language processing.

2.1 Artificial Intelligence (AI)

Artificial intelligence is a broad term that encompasses multiple scientific and engineering disciplines. The exact definition of AI remains elusive, as it has changed multiple times over the years. A prevalent concept of AI today has been described by *Russel and Norvig* [11] as an *intelligent agent* able to perceive its operating environment and act rationally according to the principles defining its purpose. Throughout the history of AI research, the paradigm of intelligent machines has undergone numerous evolutionary steps. This research path resulted in a wealth of mathematical and computational methods, which collectively comprise the modern study of AI. Figure 1 illustrates the relationship between the disciplines important for this thesis.

Historically, AI has had a mixed reputation. Despite the enthusiasm around it in its early years, the majority of the legacy AI methods have found limited application outside of the academic realm [11]. However, in the past decade, progress in *machine learning* has produced results that have reignited the interest of companies and investors in the business potential of this technology [12] to such an extent that it has made AI an integral part of the digital transformation trend [13], [14].

Although the potential of AI has been recognised, it has not yet become a general-purpose technology. While the big-tech giants have been actively capitalising on AI-powered innovation, the average adoption rate across industries has been rather slow [15]. The apparent reason for this is the difficulty of designing operational improvements which could demonstrate tangible and scalable benefits of AI technology compared to existing traditional approaches. For example, the authors of a case study about adopting ML at Uber [16] claimed around 10% of productivity improvement, although they omitted details about the cost of development and maintenance of this solution. Such examples illustrate that adopting AI technology requires careful studying, experimentation, and multifaceted evidence of its business value. Practice shows that the success of such an undertaking, being uncertain, is the product of deep technological insight, meticulous data-driven design, and strong software engineering skills [17], [18].

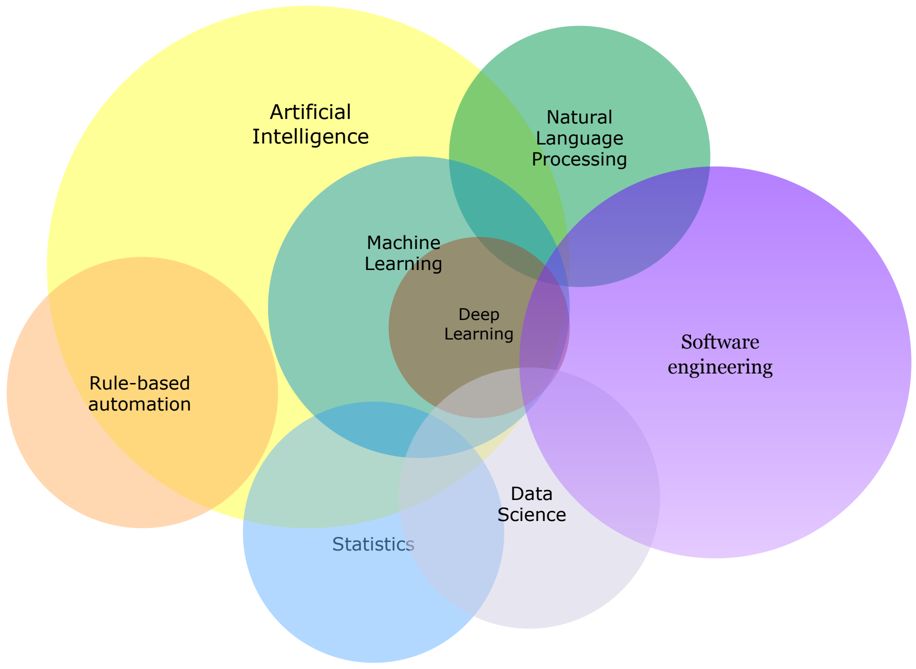


Figure 1: Venn diagram representing logical relations between the disciplines applied in this thesis. Each circle embodies a subset of methods considered in the scope of this work.

2.2 Rule-based AI

From the early years of AI and for the most of its history, rule-based methods have been the primary approach for intelligent systems design [11]. Such systems have also been referred to as *expert systems*, thus emphasising their purpose for encapsulating the knowledge of human experts in order to perform a sophisticated task [19]. Since this thesis considers a rule-based system as a point of reference for a competing approach, it is important to clarify the meaning of the term in the context of this study.

Generally, the term *rule-based AI* refers to a class of AI solutions whose underlying methods rely on pre-programmed *knowledge* for problem solving [11]. This definition agrees with the canonical architecture of rule-based systems, consisting of two main components: a knowledge base and an inference engine [19]. The knowledge base is the storage of interconnected facts and formal rules, embodying expert knowledge about the operational domain of a system. The inference engine represents the control domain of a rule-based system, whose purpose is to execute the chain of reasoning using the knowledge base to achieve the intended result.

Masri et al. [19] deem the canonical architecture that entails explicit knowledge representation to be the characteristic feature of AI rule-based systems, as opposed to other automation solutions that encapsulate knowledge implicitly in procedural code. Conversely, the taxonomy used by *Russel and Norvig* [11] does not restrict rule-based AI to a particular architecture but, instead, emphasises the underlying analytical methods, which employ different strategies to utilise pre-programmed knowledge. This latter vision is also adopted in this thesis.

Although it might not hold for the entirety of the rule-based AI, this thesis also assumes that rule-based methods are not adaptive. The lack of adaptivity, in this context, refers to the inability of an intelligent agent to autonomously adjust the parameters governing its behaviour. In other words, rule-based agents are unable to learn.

2.3 Machine Learning (ML)

Machine learning is a family of AI techniques that use data to automatically infer the parameters of a mathematical model representing the latent process that generated this data in the first place [8]. This mechanism enables an intelligent agent to *adapt* to its environment using the observed examples, in contrast to rule-based approaches that rely on a set of rules hard-coded by a designer [11]. During adaptation, an intelligent agent acquires the ability to interpret input data beyond the examples it has already seen. This capacity to generalise predictions beyond training examples is the most important quality of an ML solution, serving as the core metric in its performance evaluation [20].

Numerous machine learning strategies have been developed over the history of AI [11]. This section only explains the terms and concepts essential for this thesis, placing the main focus on *supervised learning* and its underlying theory as well as briefly presenting the notion of *unsupervised learning*. Finally, the section presents artificial neural networks and the notion of deep learning, highly relevant for this work.

General setup

A typical ML problem setup necessarily involves the following components [8], [20]–[22]: a collection of *data* points, a *machine learning model*, a *performance measure*, and an *algorithm* that ties all these components together to facilitate the learning process.

Data is an essential component of ML setup, since it serves as empirical evidence of system behaviour. Each data point represents an example of the system input x and its corresponding output y. Formally, the whole dataset S with m observations can be presented as a collection of ordered pairs $S = \{(x_i, y_i) \in X \times Y : i \in \{1, ..., m\}\}$. Depending on the context, the input vector observations $x \in X$ may be called *features* or *covariates*. While inputs are often assumed to be easily obtainable, the acquisition of outputs $y \in Y$, also known as *labels*, often requires effort and expertise. When the label y is not explicitly defined, it is possible to construct one without explicit supervision but as a property emerging from the way the input data X is distributed [22].

Generally, the observations in the dataset S are assumed to be *independent, identically distributed* (i.i.d.) realisations of a random vector (x, y) sampled from an unknown joint distribution $p(x, y)$ [8]. Based on this premise, ML methods are formulated in terms of probabilistic models broadly divided into *discriminative* and *generative* models [21]. When expressed mathematically, these models often employ free parameters, thus spanning whole families, also known as *hypothesis classes* H, of possible model instances.

The task of *learning* can hence be conceptualised as the process of searching from a family of probabilistic models H for the one $h \in H$ that most accurately explains the underlying data distribution, and hence could serve as a reliable predictor $h(x) = y$ for any data point $(x, y) \sim p(x, y)$ [8]. The mechanism for predictor learning usually employs a performance measure $L(h)$ selected depending on the learning strategy.

Supervised learning

In supervised learning, the assumption is that there exists a strong association between the input and the output that can be presented as a map $h : X \mapsto Y$, which models the system behaviour [21]. The goal of an ML algorithm is to learn such a predictor model $h(x) = \hat{y}$ that captures the aforementioned association as accurately as possible.

As a learning mechanism, the supervised learning algorithms employ loss functions $l(\hat{y}, y)$ that measure how well the model $h(x) = \hat{y}$ predicts (fits) the individual data points $(x, y) \in S$. Typically, the loss functions are defined so that for $\hat{y} \approx y$, the loss $l(\hat{y}, y) \approx 0$. Hence, the learning algorithm seeks to minimise the *empirical loss function* $\hat{L}_S(h)$ defined in [21] as

$$\hat{L}_S(h) = \frac{1}{|S|} \sum_{(x,y) \in S} l(h(x), y), \tag{1}$$

whereas the predictor that yields the best fit for the data in S by minimising the empirical loss is defined as follows [20]:

$$h_S^{ERM} = \arg\min_{h \in H} \hat{L}_S(h). \tag{2}$$

Equation (2) defines the learning rule for acquiring the optimal predictor h_S^{ERM} in its class H given the information in sample S. This learning approach is known as *empirical risk minimisation* (ERM), and the minimisation procedure executed by the learning algorithm, is referred to as *training* [20].

Although, by definition, h_S^{ERM} is the best predictor on S, it is important to bear in mind that sample S is only an approximation for the underlying distribution $p(x, y)$. This implies that the empirical loss $\hat{L}_S(h)$ is poised to vary depending on the particular realisation of S. For this reason, it is more appropriate to think about loss $l(h(x), y)$ as a random variable related to the joint distribution of the observations $p(x, y)$ [21]. Therefore, the true performance of a predictor $h(x) = \hat{y}$ should be assessed based on its average performance across all data points in the population $(x, y) \sim p(x, y)$. This idea gives premise to an important concept of *generalisation loss* expressed by the following equation [21]:

$$L_{p_{xy}}(h) = \mathbb{E}_{(x,y) \sim p_{xy}}[l(h(x), y)]. \tag{3}$$

Equation (3) provides a theoretical measure for predictor generalisation capacity, that is how inaccurately predictor h is expected to perform on average across the entire population. The notion of generalisation loss permits formulating a criterion for the optimal predictor $h^*(x) = \hat{y}^*$ minimising prediction error over every input $x \in X$. Such an optimal predictor is known as the *Bayes estimator*, formally defined in [20] as

$$h^*(x) = \arg\max_{y \in Y} \mathbb{P}(y \mid x), \tag{4}$$

which yields the prediction $\hat{y}^*$ minimising the generalisation loss over every point (x, y), as shown in the following equation [21]:

$$\hat{y}^*(x) = \arg\min_{\hat{y}} \mathbb{E}_{y \sim p_{y|x}}[l(\hat{y}, y) \mid x]. \tag{5}$$

Notice that the optimisation problem (5) involves the conditional distribution of the system output $p(y \mid x)$, which is generally unknown. Hence, the true quality of a predictor largely depends on how accurately sample S approximates $p(y \mid x)$. Moreover, there is no guarantee that the selected hypothesis class H contains the optimal Bayes estimator h^*. Consequently, as shown in the following equation [20]:

$$L_{p_{xy}}(h) - L_{p_{xy}}(h^*) = \underbrace{\left(\inf_{h \in H} L_{p_{xy}}(h) - L_{p_{xy}}(h^*)\right)}_{\text{approximation error } \epsilon_a} + \underbrace{\left(L_{p_{xy}}(h) - \inf_{h \in H} L_{p_{xy}}(h)\right)}_{\text{estimation error } \epsilon_e}, \tag{6}$$

there are two error terms that keep the generalisation loss $L_{p_{xy}}(h)$ of a predictor h away from achieving the optimal result $L_{p_{xy}}(h^*)$. First, the *approximation error* ϵ_a due to an intrinsic bias of the selected class H. Second, the *estimation error* ϵ_e reflecting the data distribution bias implied by the dataset S, which results in a predictor suboptimal in its class [20].

Tackling the approximation error ϵ_a requires selecting the hypothesis class H, so-called *inductive bias*, that most accurately encapsulates the prior information about the data distribution [8]. This is a challenging problem that involves careful data analysis and practical experimentation with the hypothesis classes of different expressive power [20]. The estimation error ϵ_e, on the other hand, is the outcome of the learning process and hence can be managed with the help of statistical learning theory. In practice, there is always a trade-off between an acceptable estimation error tolerance $\epsilon_e > 0$ and the risk of not satisfying this tolerance due to a chance $\delta \in (0, 1]$ of having a non-representative sample. This idea is at the core of the *PAC learning framework*, which with probability $1 - \delta$ produces a predictor $h \in H$ suffering estimation error no more than ϵ_e provided that an i.i.d. sample of size m is large enough and the selected hypothesis class H is *PAC learnable* [8].

Multinomial classification

When the sample space of the output variable $y \in Y$ is discrete and finite, the predictor function $h : X \mapsto Y$ is referred to as *classifier* and when the cardinality of the output set $|Y| = k > 2$, such a predictor is known as a *multinomial* (or *multiclass*) classifier [20]. As per the canonical machine learning setup, classification algorithms require a set of examples $\{(x, y) \in X \times Y\}$ to learn from, a hypothesis class of discriminator functions $H \ni h$ and a loss function $l(h(x), y)$ to minimise during training.

Although there is no shortage of *binary* classification methods, relatively few of these generalise easily to the multinomial setup described above. For this reason, it is more common to divide a single multinomial classification problem into multiple binary classification problems [8]. For example, the *aggregated methods* rely on the majority vote of multiple binary classifiers using either one-versus-rest (OVR) or one-versus-one (OVO) strategy to discriminate between the pairs of classes at a time. A more sophisticated example of an ensemble of binary classifiers adopted for the multiclass case is a family of *boosting* algorithms [20], though these remain outside of the scope of this work.

A great variety of ML models have been developed for classification. These models are based on different underlying theoretical principles, with their application-specific strengths and weaknesses as well as their own set of hyper-parameters to tune. Since each model entails a certain inductive bias, choosing one should be justified by the data underlying distribution [8]. In practice, however, it is more common to choose a model family flexible enough to have a good chance to perform well on any properly prepared data. The rationale for choosing a particular classifier type for this study is discussed in more detail in Section 4.

Within the PAC learning framework, the classification problem learnability as well as its sample complexity are characterised by the *fundamental theorem of statistical learning*. Its multiclass version with relaxed learnability assumption is encapsulated in the following relation [8]:

$$C_1 \frac{N\dim(H) + \log(1/\delta)}{\epsilon_e^2} \leq m_H(\epsilon_e, \delta) \leq C_2 \frac{N\dim(H)\log(k) + \log(1/\delta)}{\epsilon_e^2}, \tag{7}$$

for some $C_1, C_2 \in \mathbb{R}_+$.

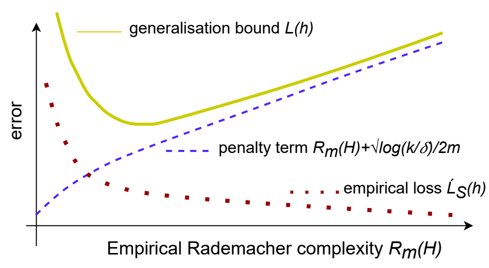
Relation (7) implies that, even with a limited sample size m (less than that of the population), it is possible to assess the performance expectations for a k-nomial classifier learned from a hypothesis class H as long as its complexity, expressed in terms of *Natarajan dimension* $N\dim(H)$, is finite.

ML predictor selection

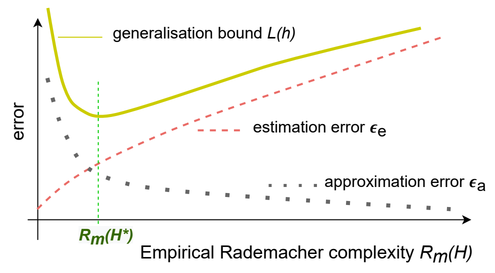
Having a good predictor h learned on a sample S using the ERM rule (2) does not guarantee the same level of performance on a different sample taken from the same population because there is always a risk that the training sample S provides an inaccurate representation of the underlying distribution $p(x, y)$ [8]. The statistical learning framework captures the relationship between this risk δ and a multinomial classifier performance expressed in terms of its generalisation loss $L_{p_{xy}}(h)$ in the following inequality [20]:

$$L_{p_{xy}}(h) \leq \hat{L}_S(h) + \frac{4k}{\rho}\mathcal{R}_m(H) + \sqrt{\frac{\log(1/\delta)}{2m}}, \tag{8}$$

where $\rho > 0$ is the margin parameter.



(a) Generalisation bound components versus predictor complexity



(b) Generalisation error terms versus predictor complexity

Figure 2: The figures adopted from [20] present the effect of empirical complexity on the generalisation error of a predictor. The generalisation bound cannot be decreased indefinitely due to the bias-complexity trade-off.

For a predictor $h \in H$ with empirical complexity $\mathcal{R}_m(H)$, trained on a sample S of size m, and accrued empirical loss of $\hat{L}_S(h)$, the relation (8) approximates the theoretical upper bound for generalisation loss $L_{p_{xy}}(h)$ with certainty of $1 - \delta$. The term $\mathcal{R}_m(H)$, known as the *Rademacher complexity*, can be thought of as an empirical measure for the proclivity of a predictor learned from the hypothesis class H to fit random noise [20].

Bias-complexity trade-off

Evidently, the best performing predictor is the one minimising all terms on the right-hand side of the relation (8). The empirical loss term $\hat{L}_S(h)$ is minimised by the ERM rule (2). The uncertainty-quantifying term $\sqrt{\log(1/\delta)/2m}$ can be addressed by increasing the sample size m at some fixed risk tolerance δ. However, managing the empirical complexity term $\mathcal{R}_m(H)$ is more difficult because of its intrinsic connection with the empirical loss term. In order to minimise empirical loss $\hat{L}_S(h)$, training algorithms require the hypothesis class H to be sufficiently expressive. Coincidentally, a more expressive hypothesis class necessarily causes the $\mathcal{R}_m(H)$ term to increase [20]. Hence, it is impossible to decrease $\hat{L}_S(h)$ without increasing $\mathcal{R}_m(H)$. This conundrum is known as the *bias-complexity trade-off* [8].

The bias-complexity trade-off manifests itself in a peculiar phenomenon consistently observed in ML practice. Although a predictor h learned from a more expressive class H usually helps to reduce its empirical loss $\hat{L}_S(h)$, the generalisation bound $L_{p_{xy}}(h)$ tends to increase after the complexity of hypothesis class $\mathcal{R}_m(H)$ exceeds a certain threshold [8]. Figure 2(a) illustrates this behaviour.

The explanation for the discussed trade-off lies in the intrinsic connection of the generalisation bound terms of relation (8) with the generalisation error components ϵ_a and ϵ_e defined in (6). The increase in predictor complexity $\mathcal{R}_m(H)$ helps reducing the empirical loss $\hat{L}_S(h)$ via a decrease in the approximation error term ϵ_a, while the estimation error term ϵ_e is poised to increase as per the relation (7). As illustrated in Figure 2(b), at the point marked as $\mathcal{R}_m(H^*)$, the ϵ_e term increases faster than ϵ_a decreases. This marks the inflection point in the upper bound of the generalisation error, after which the increase in complexity yields no performance benefit and even weakens the generalisation capacity of a predictor.

The situation when the learned predictor class complexity exceeds that reasonable threshold $\mathcal{R} > \mathcal{R}_m(H^*)$ is referred to as *overfitting*. The opposite situation, when predictor falls short of the expressive power $\mathcal{R} < \mathcal{R}_m(H^*)$ required to improve its generalisation capacity, is known as *underfitting*. Clearly, both situations are undesirable, which implies that learning a good predictor with respect to the bias-complexity trade-off amounts to finding a hypothesis class H whose complexity $\mathcal{R}_m(H)$ is as close as possible to $\mathcal{R}_m(H^*)$ [20].

The insight presented above is the main rationale for the predictor model selection used in this thesis. Numerous methods have been developed to attain a good predictor in terms of the bias-complexity trade-off. These methods are discussed in Section 4.

Unsupervised learning

Even though the input data x is usually abundant, the structural characteristics of its input space X as well as the data-generating process are often not obvious. Yet, this insight might prove useful in many ways, as even the supervised learning models are generally biased towards a particular structure of the input space [20]. Because the input data does not possess explicit clues for interpretation, there exist many ways to infer its structural properties by applying various criteria for analysis. The study of *unsupervised learning* forms a branch of machine learning techniques that emulate data-generating mechanisms to predict these structural properties and discover latent structure in complex data distributions [21].

The unsupervised learning approach has many practical applications. For example, popular methods, such as PCA and SVD, are frequently used for reducing the input data dimension [22], which also lends such high-dimensional data to visualisation.

Another important application of unsupervised learning is *feature engineering*. That is, using the information encapsulated in the raw input data to transform this data into an alternative representation that is more favourable for the task at hand [23]. An important example of feature engineering with text is transforming words into numerical embeddings that capture semantic properties of these words in an unsupervised fashion based on their co-location frequency across the training corpus [24].

In the application to text data, *topic modelling* unsupervised ML methods are designed to automatically extract sets of related themes based on the co-occurrence of words in a collection of documents [24]. The most prominent of such topic modelling methods often mentioned in the literature are *latent semantic indexing* (LSI) and *latent Dirichlet allocation* (LDA).

As a learning mechanism, unsupervised algorithms often employ similarity/distance functions $d(x_i, x_j)$ that measure the magnitude of association/difference between data points (x_i, x_j) [22]. If a method involves data transformation, it is also likely to employ a task-specific loss function to formulate an optimality condition. The exact setup varies depending on the method. This study will present the details of relevant unsupervised learning algorithms while describing the used methods in Section 4.

Deep learning

As follows from the preceding discourse, the expressive power of a machine learning model is constrained by its hypothesis class H embodying its inductive bias. This constraint is the primary cause of the approximation error ϵ_a component in (6) that caps the generalisation capacity of a predictor $h \in H$ [20]. To achieve the full learning potential, the predictor needs the expressive power that would enable it to fit the data distribution of any arbitrary complexity. Such an expressiveness is a special quality of machine learning models, known as *artificial neural networks* (ANN) [8].

The building block of any neural network is an artificial neuron, a computational unit that performs a transformation of a weighted sum of its inputs [8]. A special arrangement of these neurons into a graph-like structure of successive layers is what forms an artificial neural network. In theory, such a structure can be scaled to achieve the expressive power sufficient to represent a mathematical function of any arbitrary complexity [25].

Scalable expressive power is not the only distinguishing characteristic of ANNs. From the multilayered structure of neural networks emerges the property of abstract numerical representation of the input information [26]. This property makes ANNs fundamentally different from the conventional learning machines that are predominantly designed to solve numerical optimisation problems related to the downstream tasks, whereas the challenge of feature engineering largely remains the manual effort. In contrast, although optimisation is an essential model component, the ability of ANNs to generate abstract representations makes the tedious feature engineering task an integral part of the learning process [26].

The notion of *deep learning* (DL) hence refers to the concept of a multilayered ANN that facilitates a chain of transformations deliberately designed to yield a meaningful representation of the model input [26]. That is, the main interest of DL is an automatic learning of abstract numerical features that are effective for the learning task at hand. The 'depth' of a deep ANN is determined by the number of successive layers in its architecture [25].

Although deep neural networks have been highly successful, their benefits come at a high computational cost [25]. Training of a large deep ANN is a time-consuming process, requiring expensive hardware. In addition, the high expressive power of large ANNs inevitably increases the risk of overfitting [20]. Hence, for effective learning, deep ANNs also need a large number of training examples, which might be a blocker in some applications.

2.4 Natural Language Processing (NLP)

NLP is a branch of AI concerned with text data and intimately related to linguistics [27]. Unlike regular numerical input, text data has a more complex structure because its components (e.g., words, collocations, sentences) might have different interpretations depending on the context, not to mention other subtle qualities such as tone and sentiment that also contribute to meaning. This aspect makes text data modelling particularly challenging, since a proper mathematical representation of text should capture the context and retain information about the plausibility of word collocations [28].

The domain of NLP study is immense. This section covers only the essential NLP topics necessary to follow the discourse in the following chapters of this thesis. Particularly, the problem of language modelling is central for a variety of NLP tasks, including text encoding and text classification, that are in the focus of this study.

Statistical language modelling

Throughout the history of NLP, many sophisticated techniques have been developed to construct a formal model of natural language. Many of these techniques could be considered an example of rule-based AI, such as tagging words with their respective parts of speech and inferring the syntactic structure of sentences to extract information [29].

As for more subtle aspects of language in which context plays a decisive role, the task of *statistical language modelling* (SLM) has been traditionally used to construct probabilistic models for a word to occur alongside other words [29]. Formally, for each word w in the corpus vocabulary V, its probability to continue a sequence of length N is defined as follows [28]:

$$\mathbb{P}(w_n \mid w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n-1}, w_n)}{C(w_{n-N+1:n-1})}, \tag{9}$$

where the terms $C(\bullet)$ represent counts of the respective word sequences.

The sought probabilities defined in (9) are obtained by computing the relative frequency of the word sequences in the corpus, thus representing the maximum likelihood estimate (MLE) for the word w_n to continue the sequence $w_{n-N+1:n-1}$. Although effective, this approach suffers from numerous irredeemable drawbacks. *Goldberg* [28] highlights three primary issues that limit SLM predictive capacity. First, the rigid nature of the MLE-based approach impedes using more flexible conditioning on the context or including other relevant information that could potentially aid SLM performance. Second, scaling up the context length N in such a rigid framework results in extremely sparse distributions as N becomes larger, which is costly in terms of memory. Finally, MLE-based models do not generalise the contextual information to the previously unseen examples.

Neural language modelling

As it turns out, artificial neural networks possess the structure and capacity to overcome some of the problems of SLM [28]. That is, a neural network can be used as an alternative approach to estimating the probability distribution of a word given its context more effectively and efficiently than SLM.

In this approach, the sequence representing the context $w_{n-N+1:n-1}$ is encoded in the form of a long numerical vector x that is fed into an ANN consisting of $l \geq 1$ hidden layers and the softmax output layer that generates a probability distribution for the word w_n over the corpus vocabulary V. In the case of the feed-forward architecture, the whole forward pass operation can be represented with the following equation:

$$\mathbf{p}_V(w_n \mid w_{n-N+1:n-1}) = softmax[(g_l \circ ... \circ g_1)(x)], \tag{10}$$

where $(g_l \circ ... \circ g_1)(x)$ is the chain of composite vector functions representing the input x transformations across l network hidden layers prior to the softmax output layer, and $\mathbf{p}_V(\bullet)$ is the probability mass distribution over the corpus vocabulary V.

The setup described above forms the basis of any *neural language model* (NLM), which presents the following advantages [28]: NLM is flexible in terms of input structure, it allows an increase in context size N at a much lower cost compared to SLM, and it automatically incorporates the context information, thus being able to estimate reasonable probability distributions for the previously unseen collocation examples if the context is otherwise familiar to the model.

In addition, the compact numerical representations learned in the course of model training can encode the semantic features of every word in the corpus vocabulary [24], placing semantically related words close to each other in vector space. This property has inspired a variety of word and document embedding methods [30]–[33] that have largely contributed to the success of NLMs in NLP.

Neural language models have undergone a considerable evolution in the past decade, during which various architectures have been proposed to improve the quality of contextual representation of text as well as to tackle scalability issues of early NLMs [10]. This study is particularly interested in the latest advancement on this front, namely the recently proposed *self-attention transformer* neural network architecture, which will be presented in Section 3.2.

3 Related Work

This chapter presents the literature review on the topics that are closely related to the application domain in the focus of this thesis. Section 3.1 surveys the publications related to software log analysis and classification, emphasising the concepts especially relevant to this study. Section 3.2 gives an overview of the self-attention transformer models, their use in the transfer learning practice for NLP applications as well as the strategies used to pre-train and adapt these models for a downstream task in a specific application domain.

3.1 Software Log Analysis and Classification

Multiple research groups from the academy and industry addressed the problem of analysis and classification of log messages generated by software systems. Their publications give valuable insight into the problem anatomy and provide references for a variety of approaches to tackle challenges that arise in different application settings.

Broadly, it is possible to split the domain of log analysis research into two main focus areas: *data structuring* and *inference tasks*. This section gives a brief overview of these areas. Figure 3 summarises the related work topics identified in the course of literature review and their respective relevance to the subject of this thesis.

Log mining and data structuring

The first major research area is concerned with various forms of log mining and managing vast collections of unlabelled log data by parsing, encoding, clustering and presenting these in a structured manner. This area appears to be dominated by the unsupervised ML methods [3], [34], [35], although there were studies that attempted more rigid AI techniques that employed manual log format modelling with subsequent decomposition into tree structure and applying handcrafted mind maps for various inference tasks [36], [37].

The task of *log parsing* often employs a combination of AI approaches, including frequency analysis, clustering and various heuristic rules to infer the intrinsic log structure and convert the unstructured text into an organised table of features [34], [38]. Unfortunately, automatic parsing is rarely perfectly accurate even if logs are generated according to a consistent set of rules [3], and hence might be impractical if no structure is present in the first place. Nevertheless, log parsing results could serve as an indicator of structural features in a log corpus and provide additional clues for other analysis techniques.

Section 2.4 suggested that context in text data can be seen as an additional source of information. This aspect motivated some researchers [38]–[40] to utilise the semantic component of software logs for developing more abstract and compact encoding schemes, namely log embeddings. Not only do semantic-aware embeddings provide alternative encoding methods, but, no less important, a compact yet accurate information representation is instrumental for deep learning methods [28].

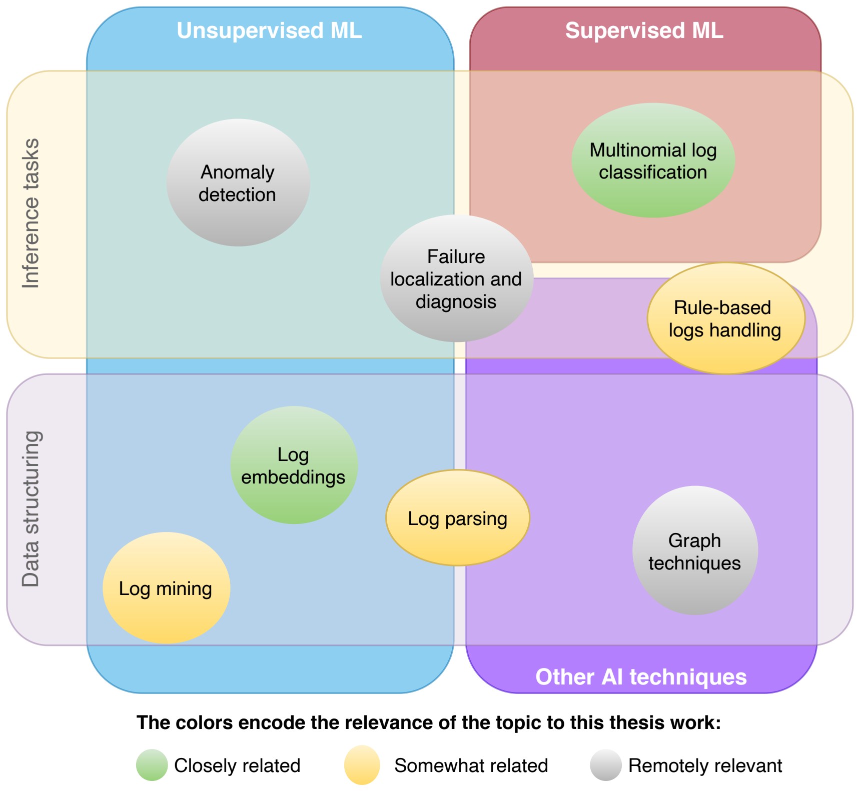


Figure 3: The map of the software log related research topics identified in the course of literature review. The topics are placed to match the underlying AI techniques and the corresponding research area.

Inference tasks

Many achievements in log data structuring and representation underlie the second large focus area of research concerned with inference tasks. The topic of *anomaly detection* enjoys particularly many publications. Several of these publications are noteworthy as application examples of neural network architectures especially effective for learning from text data, such as LSTM [7] and adaptive universal transformers [41].

The subject of *supervised multinomial log classification* is of particular interest for this work. Although it is possible to find examples of log classification in general [6], [42], [43], relatively few studies addressed the problem with the consideration of a peculiar language and the inherent structure of software logs. Among these few, LogClass [5] is a prominent publication that proposes a structure-aware log encoding scheme to improve the performance of a multinomial software log classifier. The source code of the LogClass solution has been shared for open access and may serve this work as a reference for the baseline solution.

The majority of the log classification examples found in the literature employ classical feature engineering and classification methods, implementing each data processing step manually in the form of an NLP pipeline. A rare exception in this row is one study [44] using a *convolutional neural network* for the log classification task. In addition, many recent publications have been dedicated to adaptation of neural language models for text classification, though neither of these have yet examined the domain of software logs. The concept of neural language model domain adaptation is explained in more detail in Section 3.2.

Other related work

It is also worth mentioning a few notable studies that examined other important aspects, such as the deployment of a software log classifier in the enterprise [6], [17] and use of log data for faulty component localisation [45]. These topics might be relevant to the follow-up studies succeeding this work.

3.2 Transformers in Transfer Learning

This section explains the concept of transfer learning. The section also introduces the self-attention transformer architecture and its application in neural language modelling. In addition, this section discusses the concepts of pre-training and domain adaptation of a neural language model for a downstream task.

Transfer learning

According to the theory presented in Section 2.3, the quality of a learned predictor depends on a sufficient number of diverse examples in a data sample $S = \{(x, y) \in X \times Y\}$ that serves as a proxy for the data underlying distribution $p(x, y)$ [8]. However, it is often difficult to collect sufficient data to train a good predictor, especially if its hypothesis class H is very expressive. Because training from scratch is so expensive, researchers have developed an approach, known as *transfer learning* (TL), that permits reusing knowledge about target distribution $p(x, y)$ acquired by an ML model from alternative data sources, thus facilitating effective learning even in the absence of sufficient examples to recreate this essential knowledge anew [46].

In essence, the TL approach attempts to generalise the knowledge of another predictor to the operational domain of the target application using a limited amount of domain-specific data [47]. The success of knowledge transfer depends on the similarity between the data underlying distribution of the source and the target domains as well as many other application-specific factors [46]. Over the years, dozens of variations of TL methods have been proposed to fit various application-specific circumstances. *Zhuang et al.* [46] offer a comprehensive survey on the topic as well as provide insightful examples of using the TL approach.

In recent years, transfer learning has become a popular field of study in its own right, especially for deep learning applications. It has become very successful in multiple domains, including computer vision and NLP. This thesis seeks to assess the feasibility of the TL approach to utilise the knowledge of pre-trained neural language models in order to adapt them for the software log classification task.

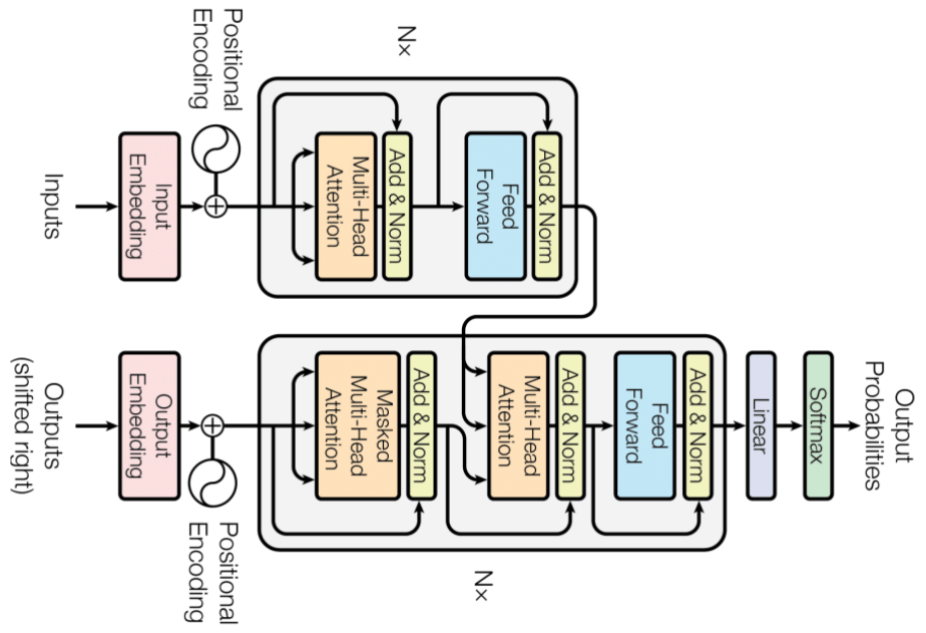


Figure 4: Self-attention transformer architecture diagram adopted from [50]. The input sequence is passed through the encoder. The encoder output is fed to the decoder that generates the output sequence in multiple passes.

Self-attention Transformer

In the past decade, the most notable advancements in NLP have been achieved with the help of deep artificial neural networks [10]. The first important breakthrough happened in 2010 when the Recurrent Neural Network (RNN) was introduced for language modelling [48]. In 2012, the LSTM cell has been proposed [49] for RNN to improve its context modelling features. Although the RNN-based models have been state-of-the-art in NLP, the structure of RNN incurs a high computational cost, thus making it slow and expensive to experiment with. A radical change in approach came in 2017 with the introduction of the *self-attention transformer* architecture [50] that has been based entirely on the *attention* mechanism and has abandoned the temporal structure of RNN in favour of a rigid structure with fixed-length input. The novel transformer architecture has enabled relatively lightweight models that, to this day, achieve state-of-the-art performance in many NLP tasks [51].

The original transformer architecture is presented in Figure 4 as it was envisioned by its creators *Vaswani et al.* [50]. It represents a sequence-to-sequence model featuring the encoder-decoder structure. The input sequence is passed through the encoder all at once to form contextual representations for each word in the context of other words in the sequence. This is achieved by means of *positional encoding* in combination with *self-attention* mechanism used throughout all encoder and decoder layers. The output sequence is produced one word at a time by selecting the most likely candidate based on the output of the decoder softmax layer. This way, generating the whole output sequence requires multiple passes through the decoder, where each subsequent pass utilises the information from the previous passes alongside the output of the encoder stack.

The primary motivation behind the transformer architecture has been to tackle three important limitations of its predecessors [50]: high computational cost, poor parallelism of computations, and the inability to keep track of long-range context dependencies in a sequence. The use of self-attention layers dramatically reduces the computational complexity as well as increases opportunities for parallelism. Even more importantly, processing the input sequence all at once enables the attention mechanism to retrieve context information from any part of the sequence within the desired range. This property is deemed the primary reason for the superior language modelling performance of transformer-based models over the previous generation of ANNs, such as RNNs and LSTM-RNNs [52].

The transformer is a deep neural network. Hence, both the encoder and decoder modules consist of multiple layers of similar blocks stacked on top of one another. The technical details about the internal composition of these blocks as well as the description of their operating principles are omitted, as these present little relevance for the following discourse. Besides, the original architecture presented in Figure 4 has undergone many variations in recent years. An interested reader can find more technical details about different variations of transformers in a survey by *Lin et al.* [52].

Another remarkable transformer quality is that its training capacity can be increased by a relatively straightforward upscaling [53]. That is, if the model performance saturates, then merely increasing the number of attention layers might help achieve tangible improvements. This discovery has motivated researchers to scale up transformers to extremes, resulting in powerful models with the capacity to encode a vast amount of knowledge from large-scale datasets [52].

Transformer-based neural language models

The immense potential of the transformer has been immediately recognised by the NLP community (and beyond) [54]. Since the initial transformer publication, a plethora of prominent studies have followed, proposing various augmentations of the original architecture and reporting new state-of-the-art results in a variety of NLP tasks [52]. Naturally, the topic of language modelling using transformers has attracted much research interest, as the ability of neural language models to produce effective contextual text representations is instrumental for other NLP tasks[10].

A pivotal point in transformer-dedicated research has been the introduction of *Bidirectional Encoder Representations from Transformers* (BERT) [55] neural language model that has utilised the encoder structure of the original transformer for the *masked language modelling* task to produce contextual representations of the input words as well as of the whole sequence. BERT has set new performance records in a variety of NLP tasks, which has affirmed the advantage of the transformer architecture over its predecessors. Since BERT has been made publicly available, it has triggered a chain reaction of follow-up studies [52] that have managed to optimise BERT and even surpass its performance achievements (e.g., RoBERTa [56]).

A notable competing branch of transformer-based language models is the family of *Generative Pre-training Transformers* (GPT) [57]–[59] that utilise the decoder structure of the original transformer and take a unidirectional approach to contextual encoding. Although the achievements of GPT models are remarkable, relatively few follow-up publications have followed, perhaps because the creators of GPT kept their work proprietary for too long.

Transformer-related research has become so dynamic that new prominent results are reported on a regular basis. Besides the aforementioned BERT and GPT, other promising transformer-based language model configurations have been developed. An interested reader can refer to the relevant surveys [10], [52], [54] for more details.

Many follow-up studies have undertaken considerable research and development effort, including training their transformer model modifications from scratch on various corpora. Much of this work has been made open source in the form of software packages. As a result, a vast collection of pre-trained transformer models have become available for public access and commercial use, which has increased the NLP community engagement and has fuelled the research activity even further. A comprehensive survey by *Kalyan et al.* [54] helps to appreciate the diversity of the produced research.

To accelerate the adoption of transformer-based solutions, multiple software libraries have included transformer models into their collections, among which the most prominent is the *transformers* library by Hugging Face [51] that features the APIs of all leading transformer architecture variants available as open source. This popularisation effort has made the state-of-the-art deep learning methods approachable for a broader community and has promoted this technology for practical applications across industries. Notably, transfer learning plays a significant role in this process.

Domain adaptation using transfer learning

The original transformer architecture incurred few assumptions about the structural properties of the input data distribution [52], thus making transformer exceptionally expressive and requiring even more training examples than its predecessors. Even the optimised and task-oriented transformer variants that incorporate a stricter inductive bias into the architecture are difficult to train from scratch without a large amount of good quality data. Because the availability of sufficient data is often a blocker for training a transformer from scratch, *transfer learning* has become a natural approach to harness the pre-trained models for practical applications [54].

As discussed earlier in this section, the NLP community has been actively experimenting with transformers and has released a series of monumental models, such as BERT, GPT, and their successors, pre-trained using language modelling objectives on diverse corpora. Each such model encapsulates a vast knowledge applicable for a variety of downstream tasks across multiple domains. The notion of *domain adaptation* hence refers to a set of strategies to achieve a satisfactory performance in a downstream task with a relatively small amount of domain-specific data by utilising the knowledge of a pre-trained model. *Kalyan et al.* [54] name several different approaches commonly used in NLP to facilitate such knowledge transfer. The considerations for using a particular domain adaptation approach will be discussed in Section 4 of this thesis.

It is also important to bear in mind that domain adaptation might also be unsuccessful due to various reasons, ranging from obvious, such as the discrepancy between source and target domains, to more sophisticated, such as catastrophic forgetting [54]. Finally, even though transfer learning does usually help to achieve good performance in downstream tasks, it is unlikely to outperform a model both pre-trained and fine-tuned on a proper corpus domain, as has been shown in the SciBERT [60] study.

4 Methods

The aim of this thesis is to develop and evaluate a learning-based solution that uses a collection of software log data to produce a classifier capable of distinguishing between observation categories. Therefore, this chapter describes the methods and design choices applied for developing the solution.

Section 4.1 introduces the current rule-based solution and the software development environment used by the client as well as describes the process of data collection and labelling. Section 4.2 describes the machine learning pipeline developed for this thesis as the baseline solution. Section 4.3 presents a competing solution approach that employs the transfer learning strategy and a transformer neural network to accomplish the same task.

4.1 Application and Data Acquisition

This section describes the client's software development environment and the role of the current rule-based solution in collecting and labelling data. The section also presents a concept for integrating the learning-based solution into the existing setup.

Software development environment

Although every software production system is unique, there is a canonical structure to the organisation of continuous integration (CI) process [2]. Typically, CI environments employ a *software version control system* and a *CI server* for centralised control over software development operations. These operations necessarily include stages to build and test new software iterations [1]. A schematic representation of the software development environment used by the client organisation is presented in Figure 5.

Data flow

When a software update is committed to *version control*, the *CI server* initiates the integration process in one of the pipelines. The pipeline builds a software package and runs basic validation and verification procedures.

Eventually, the process reaches the testing stage. At this point, the CI server initiates the integration test procedure (Connection 1 in Figure 5) on one of the *test setups* via the *CI nodes hosting test automation framework*.

First, the software is loaded into a *device under test* (DuT). Next, the test framework runs a battery of tests selected for that particular setup and its DuTs. During test execution, all test framework events, including steps of the testing procedure along with the feedback data from DuT, are recorded, thus generating logs of textual and graphical information. These logs combined with relevant metadata form test reports.

The test reports are aggregated by the CI server and combined into *test run reports* for providing a summary of the integration test run. All failed test cases in the test run report summary should be investigated in order to prevent similar failures in the future.

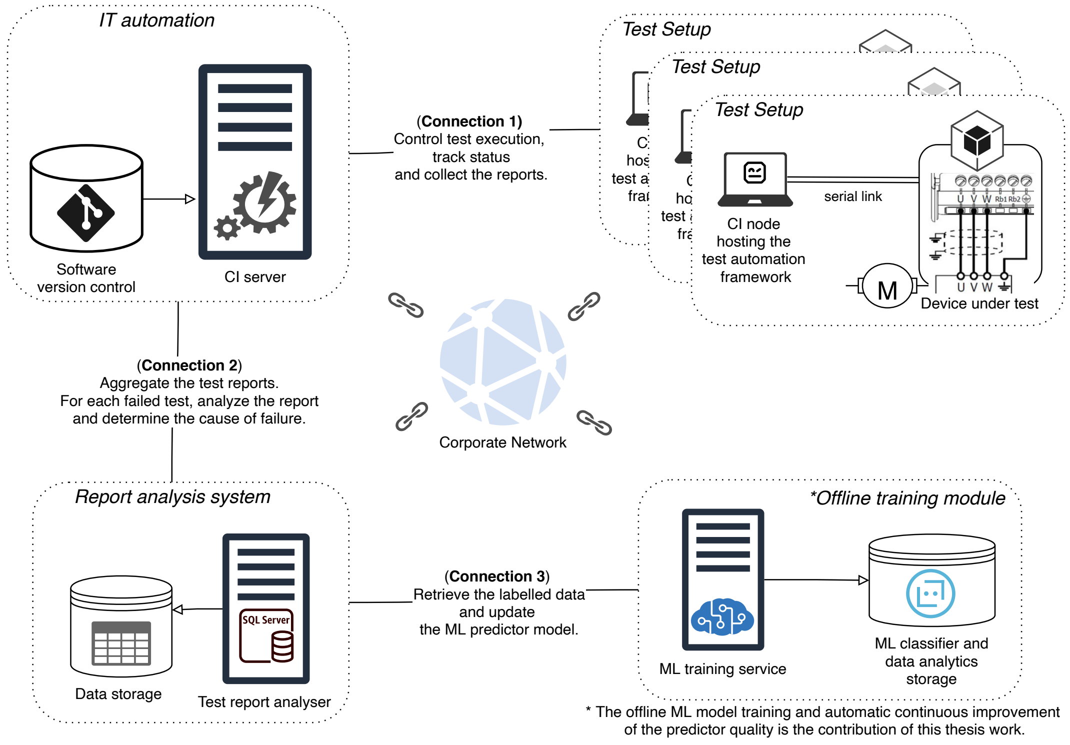

IT automation
Software version control
CI server
(Connection 1) Control test execution, track status and collect the reports.
Test Setup
Test Setup
Test Setup
CI node hosting the test automation framework
serial link
U V W Rb1 Rb2
M
U V W
Device under test
Corporate Network
(Connection 2) Aggregate the test reports. For each failed test, analyze the report and determine the cause of failure.
Report analysis system
SQL Server
Data storage
Test report analyser
(Connection 3) Retrieve the labelled data and update the ML predictor model.
*Offline training module
ML training service
ML classifier and data analytics storage
* The offline ML model training and automatic continuous improvement of the predictor quality is the contribution of this thesis work.

Figure 5: Software development environment consisting of IT automation infrastructure, test setups, report analysis system and the envisioned classifier training module. Connections illustrate data flow between the system components.

Failures are typically investigated by tracking the issue to its source, identifying its likely cause, and informing the right people about the problem. To reduce manual effort, the client's testing team developed the *test report analysis system* which uses regular expressions to match failure log text with one of the predefined failure categories. The system employs a collection of heuristic rules encapsulating the knowledge of experienced testers and a mechanism for utilising these rules for the log classification task. Notably, the client's solution description agrees well with the definition of a rule-based system given in Section 2.2.

The report analyser resides on a server of its own. It regularly collects test run reports from the CI server (Connection 2 in Figure 5), parses failed test records, and stores selected log fragments along with their metadata and automatically assigned category labels into SQL tables.

It is important to emphasise that the data used in this study has been labelled automatically by the report analyser system using the described rule-based approach. To keep the quality of automatic labelling in check and to mitigate the use of mislabelled examples, some randomly picked data samples collected for this project have been verified by experienced human testers.

Self-improving classifier

This thesis contributes to the client's failure log classification process by developing a learning-based classifier training pipeline that generates *classifier objects* for deployment in the report analysis system. As depicted in Figure 5, this pipeline could be implemented as a *training service* within the *offline training module* integrated with the rest of the software development environment via the corporate network. Using fresh user-labelled data from the report analyser (Connection 3 in the figure), this new service would routinely update and redeploy the classifier, thus facilitating automatic continuous improvement of the report analysis system.

In addition, the learning-based solutions developed in this thesis can be utilised to strengthen the existing software development environment telemetry by visualising test run summaries and providing the results of data statistical analysis implemented as an integral part of the classifier training pipeline diagnostic routines. Not only this insight is instrumental for troubleshooting a classifier, but it could also aid system architects and designers in their pursuit to improve the overall stability of the client's continuous integration pipelines.

A detailed description of the classifier training pipelines developed in this thesis are further presented in Section 4.2 and Section 4.3 of this chapter.

4.2 Baseline: Machine Learning Pipeline

The learning-based solution is envisioned as a machine learning pipeline that reads in the dataset of classification examples and outputs a classifier object that can be deployed as a component of the report analyser system presented in Section 4.1. This section explains the notion of machine learning pipeline and presents the baseline solution developed for this thesis.

The baseline solution has been implemented using *Python* [61] programming language. The solution description includes references to the major software libraries used for the implementation.

NLP classifier pipeline

The life cycle of a machine learning classifier can be divided into two phases [62]: the *build phase* when the classifier object is trained using a set of training examples, and the *operational phase* when the classifier object created during the build phase is deployed for operation in the field. This cycle repeats continuously, i.e., when a better classifier is obtained from the build phase, it replaces the currently deployed one. This thesis focuses primarily on the build phase of a machine learning pipeline with the goal of training a well-performing classifier, whereas deployment aspects are discussed only at the concept level.

The main purpose of a machine learning pipeline is to train a well-performing predictor object. Before training can even begin, ML models require the input data represented in a compatible format. In the case of text data, the words and other lexical units should be converted into numerical vectors. Furthermore, it might

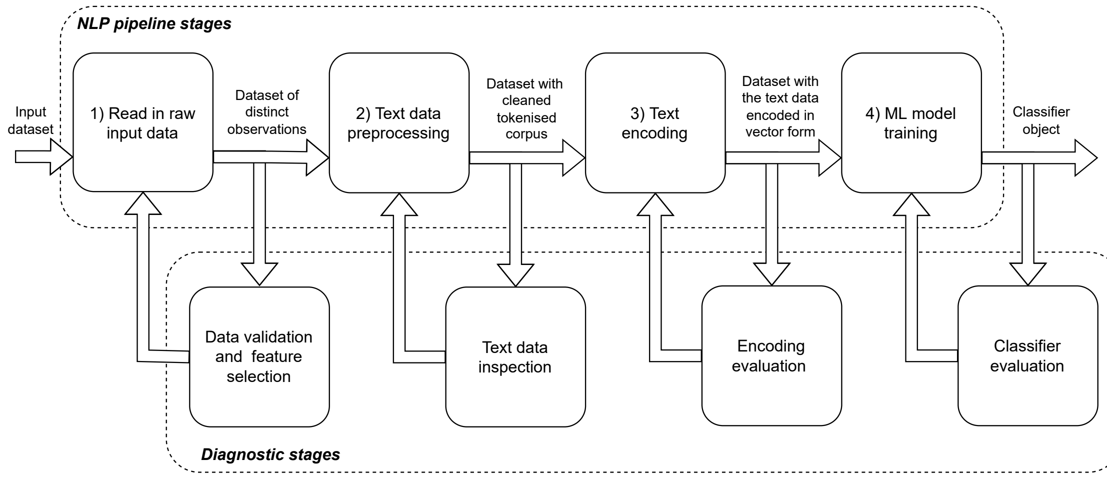

NLP pipeline stages
Input dataset
1) Read in raw input data
Dataset of distinct observations
2) Text data preprocessing
Dataset with cleaned tokenised corpus
3) Text encoding
Dataset with the text data encoded in vector form
4) ML model training
Classifier object
Data validation and feature selection
Text data inspection
Encoding evaluation
Classifier evaluation
Diagnostic stages

Figure 6: The software pipeline setup to train a log classifier object. The NLP pipeline stages perform necessary data transformations, while the diagnostic stages provide visualisation and statistical insight into each transformation step.

be desirable to filter out the noisy content that adds little value to the predictor's performance.

Figure 6 provides a schematic representation of the baseline classifier training pipeline developed for this thesis. It is divided into two main parts: *NLP pipeline stages* and *Diagnostic stages*. The former has been developed according to the canonical classification pipeline structure [4], whereas the latter has been designed with consideration for the application-specific data content. The output of each NLP pipeline stage is the result of input data transformation, which aids in training a good classifier. The results of the diagnostic stages provide feedback to a designer, i.e., generate the insight necessary to validate the outputs of the NLP pipeline stages and facilitate continuous improvement.

Stage 1. Read in raw input data

The purpose of the first stage in the NLP pipeline is to construct a proper dataset from raw input data [62]. In this case, the input data consists of failure log reports parsed and labelled by the report analyser system presented in Section 4.1.

Multiple SQL tables store different types of data about each of the log report instances. As shown on Figure 7, after the test report analyser parses a test run report, the log *text data* is stored in one table whereas the assigned *category label* for the same log instance is stored in a different table. Since there is no unique identifier to match the log instances in these tables, a custom merge key has been constructed using the unique combinations of metadata variables present in both tables.

Figure 8 illustrates the process of transforming raw input data into a proper dataset. First, SQL tables of raw input data are loaded in. The module reads and validates this input. If successful, the data passes a series of transformations to combine the log instances from separate tables into one, validate label categories, and strip the dataset of duplicate entries.

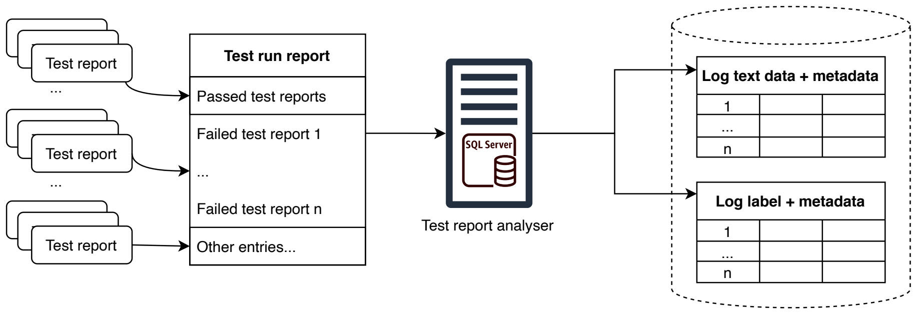


Figure 7: Data from individual test run reports processed by the test report analyser, which stores selected components of the original data into separate SQL tables, thus splitting the related parts of the log instances.

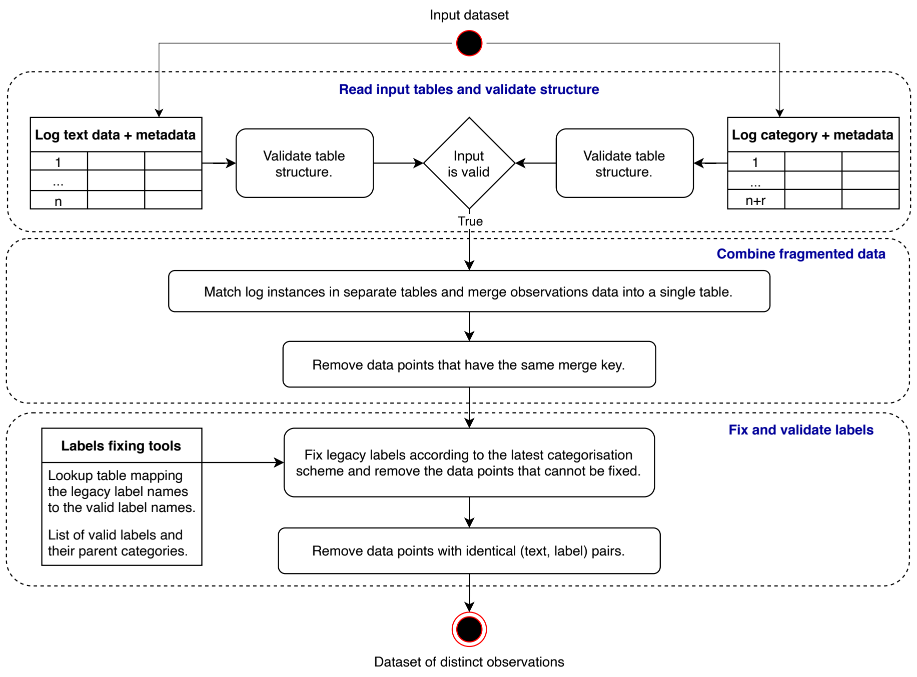

Input dataset
Read input tables and validate structure
Log text data + metadata
1
...
n
Validate table structure.
Input is valid
Validate table structure.
Log category + metadata
1
...
n+r
True
Combine fragmented data
Match log instances in separate tables and merge observations data into a single table.
Remove data points that have the same merge key.
Fix and validate labels
Labels fixing tools
Lookup table mapping the legacy label names to the valid label names.
List of valid labels and their parent categories.
Fix legacy labels according to the latest categorisation scheme and remove the data points that cannot be fixed.
Remove data points with identical (text, label) pairs.
Dataset of distinct observations

Figure 8: Decision diagram illustrating Stage 1 of the NLP pipeline. The module transforms raw input data into a proper dataset by merging two SQL tables. Preliminary cleaning includes fixing data labels and removing duplicate entries.

The outcome of the first stage is a table of distinct observations. In this thesis, a data point is considered distinct only if it contains a unique pair of *Text data* and *Label* variables. Only distinct observations make a valid set of examples for the learning algorithm.

Data validation and feature selection

Before proceeding to the next NLP pipeline stage, the result of data transformation in Stage 1 should be validated to ensure that it contains no redundant bits or missing essential information. Although certain validation procedures were implemented along with the data transformation routines of Stage 1, it is necessary to carefully examine the outcome of these transformations.

To facilitate visual inspection and statistical analysis of the resulting dataset, a diagnostic module, named *data validation and feature selection*, was developed for this thesis.

Chi-square test for feature selection

The dataset contains multiple input variables, which might either be redundant or prove useful for the classification task. The variables that have weak associations with the output can be filtered out. To assess the presence of associations between input variables and category labels, this diagnostic module employs a statistical test of independence, known as the *chi-square* χ^2 test.

The chi-square test involves computing the χ^2 statistic, which is used to test the *null hypothesis* of independence between a pair of categorical variables. The procedure consists of the following steps [63]:

1. Verify that the setup complies with the assumptions and limitations of the χ^2 test.

2. For each modality pair (i, j) between the two variables, collect the frequencies of observations o_{ij} into the bi-variate contingency table.

3. For each modality pair (i, j) between the two variables, compute the expected frequency of occurrence e_{ij} under the assumption of statistical independence.

4. Compute the χ^2 statistic value as follows:

$$\chi^2 = \sum_{i=1}^{k} \sum_{j=1}^{m} \frac{(o_{ij} - e_{ij})^2}{e_{ij}}, \tag{11}$$

   where k and m are the respective number of modalities in the two variables under test.

5. Choose the significance value α and find the critical χ^2 score from the χ^2 distribution table. If the computed χ^2 value exceeds the critical score, then the null hypothesis may be rejected.

Correspondence analysis for studying associations

Although the χ^2 test is a good indicator of possible associations between the variables, it provides little insight into the modalities that contributed to the χ^2 score. To determine the role of individual modalities in bivariate correspondence, this diagnostic module implements the statistical technique, known as *correspondence analysis*.

Assuming that the intermediate results from computing the χ^2 statistic are available, the correspondence analysis procedure consists of the following steps [64]:

1. Compute the row profile table.

2. From the row profile table, compute the χ^2 distance, mass, and inertia values for each of the row variable modalities.

3. Compute the column profile table.

4. From the column profile table, compute the χ^2 distance, mass, and inertia values for each of the column variable modalities.

5. Compute the matrix of standardised *Pearson residuals* R with elements $r_{ij} = \frac{o_{ij}-e_{ij}}{N\sqrt{e_{ij}}}$, where N is the total number of observations in the dataset.

6. Perform the *singular value decomposition* of the matrix R obtained in Step 5 to yield the following representation:

$$R = U\Sigma V^T, \tag{12}$$

where the matrices U and V hold the principal axes for the row and column variables, respectively, and Σ is the matrix of singular values, representing the standard deviations of Pearson residuals along the principal axes.

Correspondence analysis provides a means for visualising the relationship between modalities. Figure 9 shows an example of a correspondence analysis plot generated by this diagnostic module for the (*Target*, *Label*) variable pair. This plot illustrates a projection of the principal axes to a 2-dimensional subspace, preserving as much information as possible about the associations between modalities. The scope of this work does not permit discussing bivariate correspondence in detail. This example merely illustrates the availability of a powerful statistical analysis tool in the developed solution. The instructions for interpreting correspondence analysis plots can be found in [64].

Stage 1 implementation

Stage 1 was implemented as a Python module using the decision diagram in Figure 8 as a specification.

The input data tables were provided in the form of serialised Pandas [65] DataFrames. The DataFrame data structure was the primary format for managing tabular data throughout this project. Likewise, many data transformation tasks, such as constructing custom merge keys and removing duplicate entries, were performed with Pandas and NumPy [66] software libraries.

In the implementation of Stage 1 diagnostic features, Pandas and NumPy were used to construct tables and perform matrix operations. The χ^2 critical score values were obtained with the help of SciPy [67] software library. Scikit-learn [68] library has been used to implement singular value decomposition.

For data visualisation, a set of graph rendering functions was implemented using Plotly [69] graphing software library.

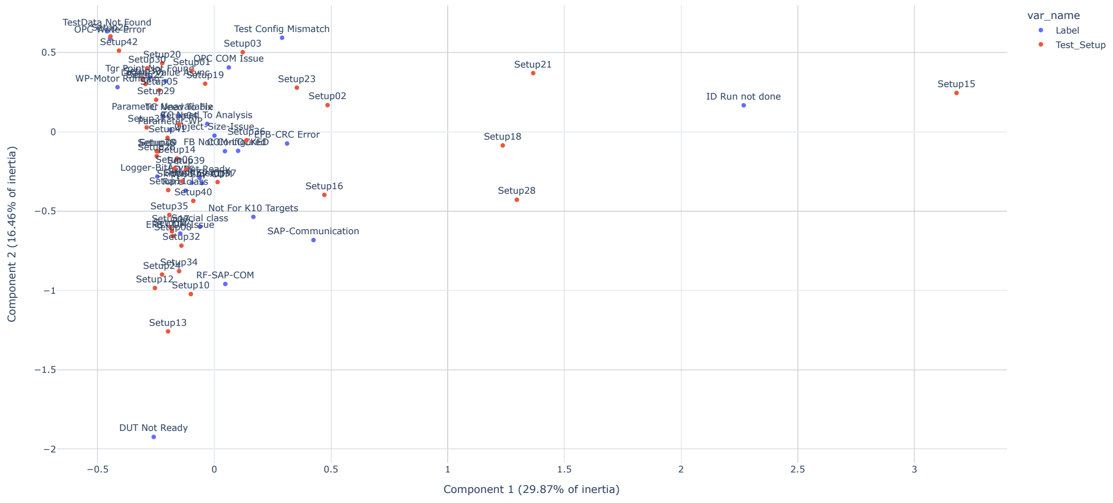


Figure 9: Correspondence analysis plot illustrating bi-variate correspondence between the variables *Target* and *Label*. The modalities are presented in terms of the two most significant principle components.

Stage 2. Text data preprocessing

The preprocessing stage is an important precursor to converting text data into a structured form suitable for multidimensional representation [24]. In this form, the individual tokens become variables used as characteristic features of every observation in the dataset. However, not all tokens are equally valuable for the learning task. Moreover, every new variable increases the memory demand for data representation and adds a computational burden to the learning algorithm [22]. Hence, the purpose of the *text data preprocessing* stage is to clean the logs from noisy and redundant content, thus preserving only those units of text that aid the learning task both in terms of accuracy and efficiency [4].

Figure 10 illustrates the data transformations performed in this stage. First, each text data sample is tokenised, i.e., split into a sequence of individual tokens. Next, a series of transformations is performed on every token of every text data sample. As a result, the text samples become shorter. Composed of only a fraction of the original vocabulary, many logs end up as duplicates, which permits retiring them from the dataset. The output of this module is a cleaned and tokenised dataset, optimised for text vectorisation to be performed at Stage 3 of the NLP pipeline.

The text preprocessing procedure involves various transformations of text data, including merging, splitting, stripping tokens of certain symbols, and entirely removing certain kinds of tokens from the corpus vocabulary. Although some preprocessing techniques, such as lowercasing and removal of stopwords, are typical for any NLP application, identifying the transformations that would achieve the optimal text data composition requires substantial prior knowledge about the application domain [24].

For this thesis, an extensive collection of preprocessing functions has been designed based on the client's input and a systematic analysis of the corpus content. Most

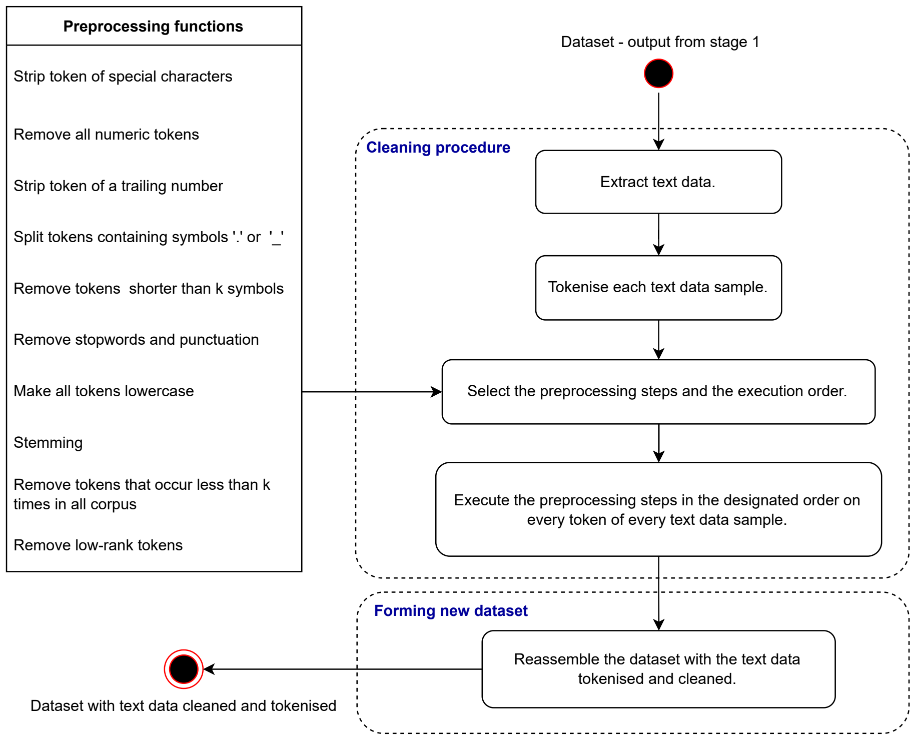

Preprocessing functions
Strip token of special characters
Remove all numeric tokens
Strip token of a trailing number
Split tokens containing symbols '.' or '_'
Remove tokens shorter than k symbols
Remove stopwords and punctuation
Make all tokens lowercase
Stemming
Remove tokens that occur less than k times in all corpus
Remove low-rank tokens
Dataset - output from stage 1
Cleaning procedure
Extract text data.
Tokenise each text data sample.
Select the preprocessing steps and the execution order.
Execute the preprocessing steps in the designated order on every token of every text data sample.
Forming new dataset
Reassemble the dataset with the text data tokenised and cleaned.
Dataset with text data cleaned and tokenised

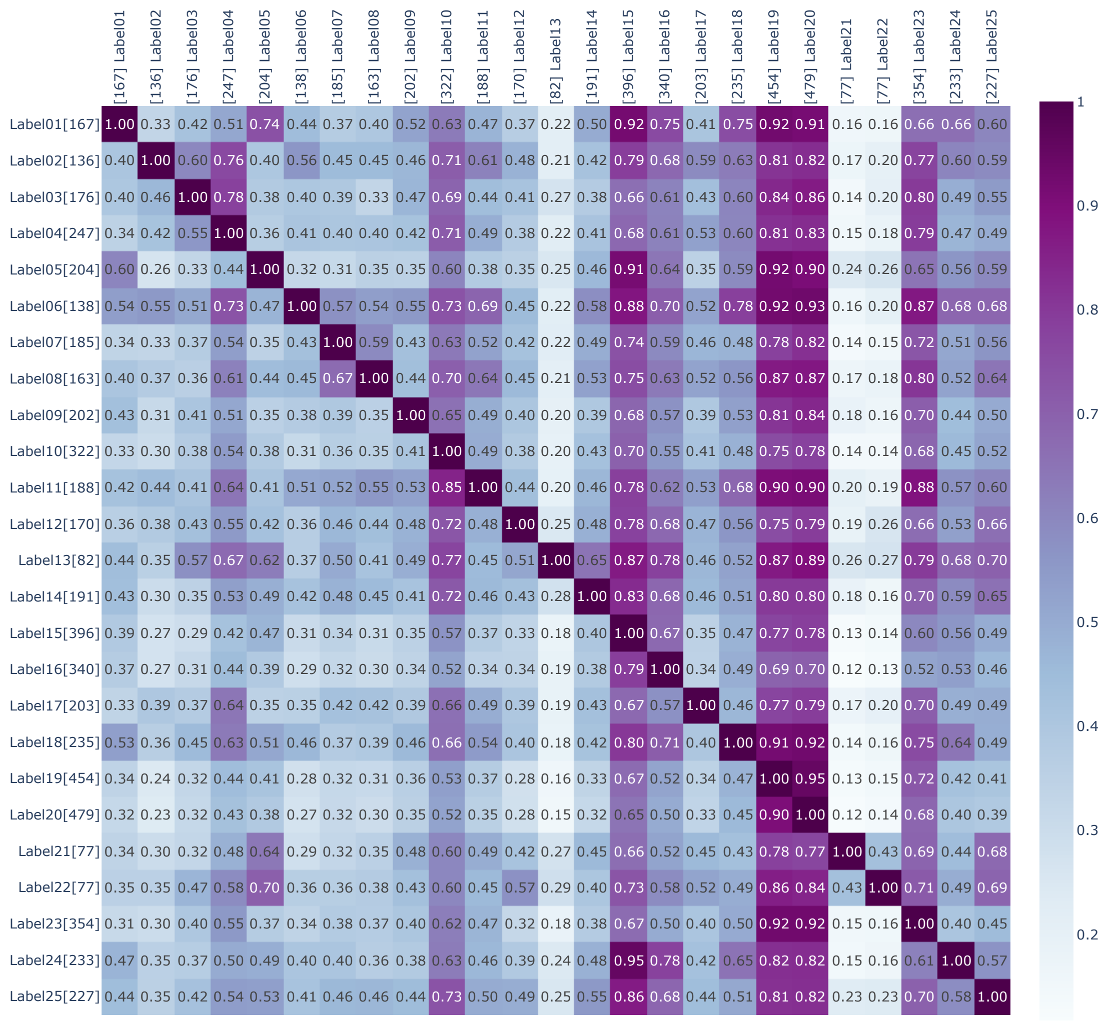
Figure 10: Decision diagram illustrating Stage 2 of the NLP pipeline. The module performs a series of preprocessing procedures on the text data to reduce noise content and optimise corpus vocabulary.

of these functions (shown in Figure 10) are rather unsophisticated and are named in a self-explanatory manner. For example, it is typical for stack traces to contain long compound tokens composed of multiple words connected using '.' or '_' symbols. Therefore, there is a preprocessing function that splits these long tokens into multiple short tokens, thus helping to optimise the log corpus vocabulary.

Stage 2 diagnostics

The main difficulty of the text preprocessing task, in general, and with the text data in this dataset, in particular, is that the logs' content is very application-specific and lacks prior structure. Although there exist techniques, such as *topic modelling* [24] and *log parsing* [3], which facilitate automatic pattern extraction from text data, neither of the attempted methods, including *latent Dirichlet allocation* [70] and *LogParse* [34], achieved results that could obviate the need for manual inspection of the text data.

For the text data analysis, the *token inspector* utility developed in this thesis provides a pervasive insight into the logs' content. Not only this tool aids in designing and improving the text preprocessing procedure, but it also helps in investigating the cases of misclassification when evaluating the model at Stage 4 of the NLP pipeline.



	[167] Label01	[136] Label02	[176] Label03	[247] Label04	[204] Label05	[138] Label06	[185] Label07	[163] Label08	[202] Label09	[322] Label10	[188] Label11	[170] Label12	[82] Label13	[191] Label14	[396] Label15	[340] Label16	[203] Label17	[235] Label18	[454] Label19	[479] Label20	[77] Label21	[77] Label22	[354] Label23	[233] Label24	[227] Label25
Label01[167]	1.00	0.33	0.42	0.51	0.74	0.44	0.37	0.40	0.52	0.63	0.47	0.37	0.22	0.50	0.92	0.75	0.41	0.75	0.92	0.91	0.16	0.16	0.66	0.66	0.60
Label02[136]	0.40	1.00	0.60	0.76	0.40	0.56	0.45	0.45	0.46	0.71	0.61	0.48	0.21	0.42	0.79	0.68	0.59	0.63	0.81	0.82	0.17	0.20	0.77	0.60	0.59
Label03[176]	0.40	0.46	1.00	0.78	0.38	0.40	0.39	0.33	0.47	0.69	0.44	0.41	0.27	0.38	0.66	0.61	0.43	0.60	0.84	0.86	0.14	0.20	0.80	0.49	0.55
Label04[247]	0.34	0.42	0.55	1.00	0.36	0.41	0.40	0.40	0.42	0.71	0.49	0.38	0.22	0.41	0.68	0.61	0.53	0.60	0.81	0.83	0.15	0.18	0.79	0.47	0.49
Label05[204]	0.60	0.26	0.33	0.44	1.00	0.32	0.31	0.35	0.35	0.60	0.38	0.35	0.25	0.46	0.91	0.64	0.35	0.59	0.92	0.90	0.24	0.26	0.65	0.56	0.59
Label06[138]	0.54	0.55	0.51	0.73	0.47	1.00	0.57	0.54	0.55	0.73	0.69	0.45	0.22	0.58	0.88	0.70	0.52	0.78	0.92	0.93	0.16	0.20	0.87	0.68	0.68
Label07[185]	0.34	0.33	0.37	0.54	0.35	0.43	1.00	0.59	0.43	0.63	0.52	0.42	0.22	0.49	0.74	0.59	0.46	0.48	0.78	0.82	0.14	0.15	0.72	0.51	0.56
Label08[163]	0.40	0.37	0.36	0.61	0.44	0.45	0.67	1.00	0.44	0.70	0.64	0.45	0.21	0.53	0.75	0.63	0.52	0.56	0.87	0.87	0.17	0.18	0.80	0.52	0.64
Label09[202]	0.43	0.31	0.41	0.51	0.35	0.38	0.39	0.35	1.00	0.65	0.49	0.40	0.20	0.39	0.68	0.57	0.39	0.53	0.81	0.84	0.18	0.16	0.70	0.44	0.50
Label10[322]	0.33	0.30	0.38	0.54	0.38	0.31	0.36	0.35	0.41	1.00	0.49	0.38	0.20	0.43	0.70	0.55	0.41	0.48	0.75	0.78	0.14	0.14	0.68	0.45	0.52
Label11[188]	0.42	0.44	0.41	0.64	0.41	0.51	0.52	0.55	0.53	0.85	1.00	0.44	0.20	0.46	0.78	0.62	0.53	0.68	0.90	0.90	0.20	0.19	0.88	0.57	0.60
Label12[170]	0.36	0.38	0.43	0.55	0.42	0.36	0.46	0.44	0.48	0.72	0.48	1.00	0.25	0.48	0.78	0.68	0.47	0.56	0.75	0.79	0.19	0.26	0.66	0.53	0.66
Label13[82]	0.44	0.35	0.57	0.67	0.62	0.37	0.50	0.41	0.49	0.77	0.45	0.51	1.00	0.65	0.87	0.78	0.46	0.52	0.87	0.89	0.26	0.27	0.79	0.68	0.70
Label14[191]	0.43	0.30	0.35	0.53	0.49	0.42	0.48	0.45	0.41	0.72	0.46	0.43	0.28	1.00	0.83	0.68	0.46	0.51	0.80	0.80	0.18	0.16	0.70	0.59	0.65
Label15[396]	0.39	0.27	0.29	0.42	0.47	0.31	0.34	0.31	0.35	0.57	0.37	0.33	0.18	0.40	1.00	0.67	0.35	0.47	0.77	0.78	0.13	0.14	0.60	0.56	0.49
Label16[340]	0.37	0.27	0.31	0.44	0.39	0.29	0.32	0.30	0.34	0.52	0.34	0.34	0.19	0.38	0.79	1.00	0.34	0.49	0.69	0.70	0.12	0.13	0.52	0.53	0.46
Label17[203]	0.33	0.39	0.37	0.64	0.35	0.35	0.42	0.42	0.39	0.66	0.49	0.39	0.19	0.43	0.67	0.57	1.00	0.46	0.77	0.79	0.17	0.20	0.70	0.49	0.49
Label18[235]	0.53	0.36	0.45	0.63	0.51	0.46	0.37	0.39	0.46	0.66	0.54	0.40	0.18	0.42	0.80	0.71	0.40	1.00	0.91	0.92	0.14	0.16	0.75	0.64	0.49
Label19[454]	0.34	0.24	0.32	0.44	0.41	0.28	0.32	0.31	0.36	0.53	0.37	0.28	0.16	0.33	0.67	0.52	0.34	0.47	1.00	0.95	0.13	0.15	0.72	0.42	0.41
Label20[479]	0.32	0.23	0.32	0.43	0.38	0.27	0.32	0.30	0.35	0.52	0.35	0.28	0.15	0.32	0.65	0.50	0.33	0.45	0.90	1.00	0.12	0.14	0.68	0.40	0.39
Label21[77]	0.34	0.30	0.32	0.48	0.64	0.29	0.32	0.35	0.48	0.60	0.49	0.42	0.27	0.45	0.66	0.52	0.45	0.43	0.78	0.77	1.00	0.43	0.69	0.44	0.68
Label22[77]	0.35	0.35	0.47	0.58	0.70	0.36	0.36	0.38	0.43	0.60	0.45	0.57	0.29	0.40	0.73	0.58	0.52	0.49	0.86	0.84	0.43	1.00	0.71	0.49	0.69
Label23[354]	0.31	0.30	0.40	0.55	0.37	0.34	0.38	0.37	0.40	0.62	0.47	0.32	0.18	0.38	0.67	0.50	0.40	0.50	0.92	0.92	0.15	0.16	1.00	0.40	0.45
Label24[233]	0.47	0.35	0.37	0.50	0.49	0.40	0.40	0.36	0.38	0.63	0.46	0.39	0.24	0.48	0.95	0.78	0.42	0.65	0.82	0.82	0.15	0.16	0.61	1.00	0.57
Label25[227]	0.44	0.35	0.42	0.54	0.53	0.41	0.46	0.46	0.44	0.73	0.50	0.49	0.25	0.55	0.86	0.68	0.44	0.51	0.81	0.82	0.23	0.23	0.70	0.58	1.00

Figure 11: Matrix of shared token ratios between each pair of 25 selected labels. The heatmap emphasises the entries for the pairs of classes that share much of the same vocabulary. The values in square brackets show the labels' vocabulary sizes.

Comparing class-specific vocabularies

Since text data is deemed the primary source of information used to distinguish between failure categories, it would be helpful to know how many common tokens these categories share. For this purpose, this diagnostic module computes a pairwise similarity metric $s : C \times C \mapsto [0, 1]$ that may serve as a proxy for separability between two classes based on their vocabulary content. The metric s is defined as follows:

$$s(c_1, c_2) = \frac{|V_{c_1} \cap V_{c_2}|}{|V_{c_1}|}, \tag{13}$$

where $V_{c_1}, V_{c_2} \subseteq V$ are the vocabularies of the respective classes $c_1, c_2 \in C$.

Figure 11 presents the *token-share matrix* S that aggregates the pairwise metrics $s(c_1, c_2)$ for all classes $c \in C$. Notice, the matrix is not symmetric. This is because each entry is the ratio of the number of tokens shared between the two classes

$|V_{c_1} \cap V_{c_2}|$ and the total number of unique tokens in the vocabulary of the class located in the matrix row $|V_{c_1}|$.

For example, the matrix entry *(Label16, Label01)=0.37* indicates that 37% of tokens in the vocabulary of the *Label16* class were also found in the vocabulary of the *Label01* class. However, the symmetric entry *(Label01, Label16) = 0.75* indicates that a much larger share of *Label01* tokens also occur in *Label16* class vocabulary, which is expected since the latter has the vocabulary twice as large than that of the former.

Tokens ranking

Since tokens are to become the learning features in the classification task, it is helpful to have a mechanism to rank them in terms of their predictive potential. For this purpose, the token inspector tool keeps record of classes associated with a particular token and implements special *ubiquity* metric.

Define matrix $\bar{U} \in \mathbb{R}^{|C|\times|V|}$, where C is the set of category labels (classes) and V is the set of unique tokens in the log corpus vocabulary. The matrix is composed as follows:

$$\bar{U} = [u_{token_1}, u_{token_2}, ..., u_t..., u_{token_{|V|}}],$$

where each $u_t \in \mathbb{R}^{|C|}$ is a column vector containing the *ubiquity weights* u_{ct} for the token $t \in V$, each weight indexed by the name of the respective class $c \in C$. Each ubiquity weight $u_{ct} \in [0, 1]$ is computed as the share of class-specific observations containing token t, formally expressed in the following equation:

$$u_{ct} = \frac{|S_{ct}|}{|S_c|}, \tag{14}$$

where $S_c = \{(x, y) \in S : y = c\}$ is the set of observations from the dataset S attributed to class c, and $S_{ct} \subseteq S_c$ is the subset of observations in S_c that contain the token t.

For example, for the token *'call'* in the log corpus vocabulary, the token inspector produces the following vector:

$$u_{call} = [\underbrace{0}_{\text{Label01}}, ..., \underbrace{0.09}_{\text{Label12}}, ..., \underbrace{0.01}_{\text{Label15}}, ..., \underbrace{0.99}_{\text{Label22}}, ..., \underbrace{0}_{\text{Label25}}]^T,$$

which contains 3 nonzero ubiquity weights for the classes that happen to have this token in their vocabularies. Based on this output, the token *'call'* is especially ubiquitous among the *Label22* class observations (99%). Furthermore, since only 3 labels contain this token in their vocabularies, the occurrence of this token in an observation narrows the set of plausible classification results to the 3 options shown in the list above.

Clearly, the fewer labels share the same token and the more ubiquitous this token is within its class, the more predictive power such token possesses relative to others. This idea is at the core of the ranking scheme developed for this thesis, which assigns a *value score* v_t to each token t in the log corpus vocabulary. The token value score is computed as follows:

$$v_t = \frac{\|u_t\|_\infty}{\|u_t\|_1} \|u_t\|_\infty \tag{15}$$

where $\|u_t\|_\infty$ yields the maximum ubiquity weight in the vector u_t, and the ratio $\|u_t\|_\infty / \|u_t\|_1$ quantifies the maximum ubiquity relative to the cumulative ubiquity of the token t across all classes.

This scoring approach permits establishing a relative ranking between the tokens in the log corpus vocabulary. Inspired by the discourse on the discriminative characteristics of statistical weighting of tokens in [27], this ranking scheme has been developed as a data-driven alternative to the heuristic method for feature selection used by the creators of the original rule-based solution. It has been used to design and validate several preprocessing steps and serves as the dimensionality reduction step in Stage 2.

There exist other approaches to ranking of variables in terms of their discriminative properties. For example, *Gini index* is often used in NLP applications [24]. However, Gini index is a relative measure, which does not reflect the true predictive power of a token. Alternatively, *information gain* [29] and *conditional entropy* [24] measures could be computed for the tokens to quantify their predictive power from the information-theoretical perspective. While certainly effective, these measures were found more difficult to interpret when analysing the reason for the given score, which might be an impediment when debugging the solution.

Visual inspection

There are many ways to examine text data. Besides computing formal metrics, the token inspector tool provides the means of visual inspection. One example is illustrated in Figure 12, in which the vocabularies of two classes are compared in terms of tokens ubiquity.

Stage 2 implementation

Stage 2 was implemented as a Python module, using the decision diagram in Figure 10 as a specification.

Text tokenisation was implemented using NLTK [71] software library. The majority of the preprocessing functions were implemented with the help of the Regex package from Python's standard library, except for the *stemming* routine, which employs sophisticated linguistic knowledge implemented in NLTK.

The Stage 2 diagnostic module was implemented with Pandas [65] and Numpy [66] software libraries.

Stage 3. Text encoding

Learning algorithms work with numbers. Therefore, all nonnumerical data, such as text, should be transformed into a numerical form. The groundwork for this

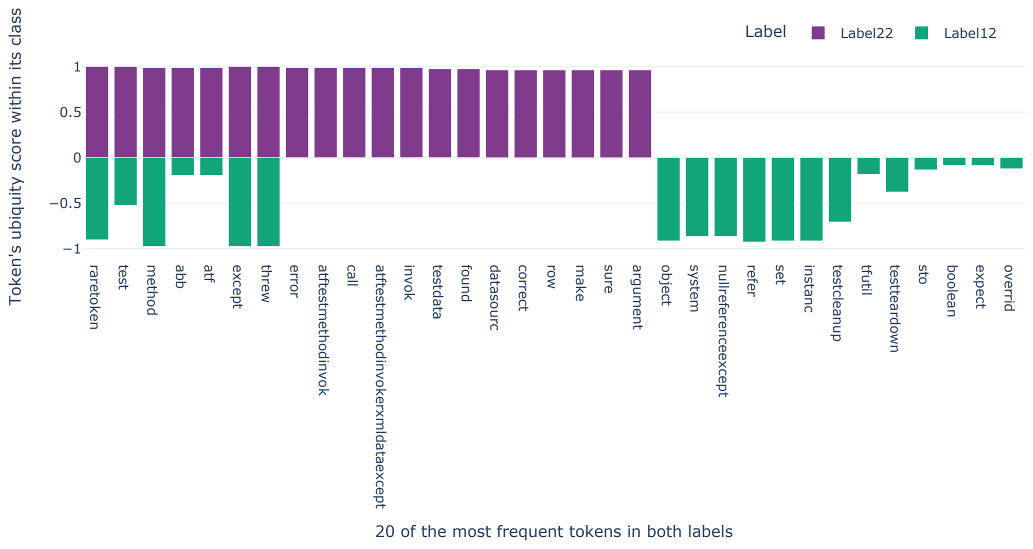


Figure 12: The bar chart stacks the ubiquity scores for the 20 most ubiquitous tokens occurring in two labels. The ubiquity scores of the label marked green have been inverted negative to facilitate pairwise comparison for the shared tokens.

step has been performed at the previous stage by extracting text features deemed most relevant for the learning task. The purpose of Stage 3 is to encode these features in such a way that their discriminative qualities propagate to the vector space representation, which is pivotal for effective learning [24].

Figure 13 presents the main steps of the encoding procedure. First, tokenised text data is extracted from the dataset. Next, the content of every log is transformed into the vector space representation according to the selected encoding scheme. The module permits selecting a number of different encoding options to obtain multiple dataset variants to experiment with. Additionally, all remaining nonnumerical variables in the dataset are converted into numerical form. Finally, this module constructs and stores a collection of dataset variants in which all observations are represented as numerical vectors. In this format, the dataset is ready for the learning task.

It is difficult to predict in advance which encoding scheme is going to yield the vector space representation most suitable for the task at hand. In NLP practice, it is common to attempt multiple encoding strategies when designing a machine learning solution [4], [62]. Therefore, in this thesis, Stage 3 generates a collection of dataset variants, each based on a different encoding scheme.

Count-based vector space representation models

A traditional approach to text vectorisation is based on counting the occurrence of individual tokens in a text sample [4]. This way, each text sample is represented as a combination of token counts stacked in a $|V|$-dimensional numerical vector. Thus, the whole corpus is represented as *document-term matrix* $D \in \mathbb{R}^{N \times |V|}$, where N is the number of observations in the dataset and $|V|$ is the number of unique tokens in

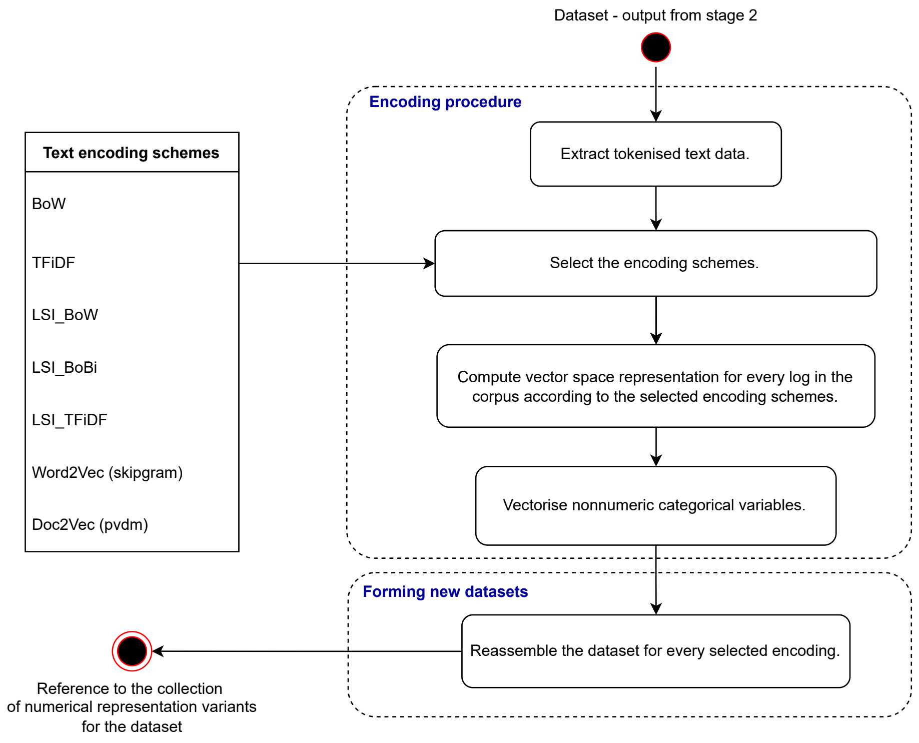

Dataset - output from stage 2
Encoding procedure
Extract tokenised text data.
Text encoding schemes
BoW
TFiDF
LSI_BoW
LSI_BoBi
LSI_TFiDF
Word2Vec (skipgram)
Doc2Vec (pvdm)
Select the encoding schemes.
Compute vector space representation for every log in the corpus according to the selected encoding schemes.
Vectorise nonnumeric categorical variables.
Forming new datasets
Reassemble the dataset for every selected encoding.
Reference to the collection of numerical representation variants for the dataset

Figure 13: The decision diagram illustrating Stage 3 of the NLP pipeline. The module transforms the nonnumerical data content of the dataset into vector-space representation using one or more of the selected encoding schemes.

the corpus vocabulary. This encoding scheme, known as *bag of words* (BoW), does not capture the context information, since the word order is not taken into account. Despite its simplicity, this representation model is an important baseline considered alongside other encoding schemes used in this thesis.

The BoW approach can be used to represent text samples in terms of pairs (or any other number) of consecutively occurring tokens. In practice, it means enlarging the vocabulary with the instances of tokens' collocations found in the corpus. On the one hand, this approach helps to encode semantic information into the vector representation [24]. On the other hand, the sheer number of rare unique collocations in the corpus often results in a document-term matrix that is extremely bulky and sparse. For this reason, the *bag of bigrams* (BoBi) representation used in this work appears in the classification task only in its dense form (LSI_BoBi), which is presented in the following subsection.

It is not unreasonable to hypothesise that the logs of a particular class contain tokens that are rare among other classes. It might be beneficial to emphasise the weights of such tokens in the vector representation, as this might help to distinguish the observations containing such tokens as common for a specific class [4]. The encoding scheme, known as *TFiDF*, implements the described weighting principle by computing two weighting terms [27]: $tf_{t,d} = \log_{10}[count(t,d) + 1]$, the token frequency t in a text sample d, and $idf_t = \log_{10}\frac{N}{df_t}$, the inverse of the number of text samples d containing token t relative to all documents in the corpus. Hence, in

TFiDF dataset, each text sample d is represented as a vector of real-valued token weights computed as the product of these two terms: $w_{t,d} = tf_{t,d} idf_t$.

In addition to classical count-based text representation methods presented above, other similar encoding schemes have been proposed to adapt to domain-specific structure of text. For example, the LogClass [5] study has proposed the TFiLF encoding method that takes into account the consistency of a token placement order across the log corpus. However, the assumptions that underlie this encoding scheme do not hold for the peculiar structure of the logs used in this study. Therefore, although it was attempted in this work, the TFiLF is not included for further evaluation along with other methods but rather appears in this work as an honorary mention.

Latent semantic indexing (LSI)

The count-based representations are simple and intuitive. However, the resulting document-term matrices tend to be sparse, which is an inefficient way to store and process information [24]. Besides, it might be the case that even after careful cleaning during the preprocessing at Stage 2, the corpus vocabulary ends up so large that the explicit vector representation based on term counts becomes too computationally costly for the learning algorithm.

The technique, known as *latent semantic indexing* (LSI) or *latent semantic analysis* (LSA) is a popular method to compress the document-term matrices into a compact dense format [24]. In practice, LSI is the NLP-adopted name for the singular value decomposition (SVD) technique, which is often used for dimensionality reduction in various applications. For a document-term matrix $D \in \mathbb{R}^{N\times|V|}$, LSI yields the following decomposition [24]:

$$D = Q\Sigma P^T \tag{16}$$

where, for $k = min\{N, |V|\}$, $Q \in \mathbb{R}^{N\times k}$ is an orthonormal matrix of eigenvectors of DD^T, $P \in \mathbb{R}^{|V|\times k}$ is an orthonormal matrix of eigenvectors of $D^T D$, and the diagonal matrix $\Sigma \in \mathbb{R}^{k\times k}$ holding the singular values ordered from largest to smallest.

In this arrangement, the rows of matrix $\Sigma P^T \in \mathbb{R}^{k\times|V|}$ form a new basis for representing every text sample of the dataset D in vector space using the rows of the matrix Q as the coordinates. Thanks to the transformation performed by LSI, the new basis permits using only a fraction of the k components held in the LSI matrices while retaining most of the original information of the matrix D. As an additional benefit, the LSI representation often makes semantically related terms and documents cluster closer to each other [24], which might aid data separability.

Static semantic embeddings

The encoding schemes described above have limited capacity for context modelling. Although LSI can be seen as a form of abstraction of the count-based representation, it is performed on top of encoding schemes that ignore word order and hence lack important semantic clues. Yet, context is an important source of information [24], which could make the numerical representation of text more effective for the downstream task. This idea underlies the category of *distributed representations*, also known as *semantic embeddings*, that use context information for word encoding [28].

In practice, the idea of context-aware word representation is closely related to the neural language modelling task presented in Section 2.4. Similarly, the distributed representation approach utilises a neural network to learn abstract numerical representations of words from their surrounding context, but it optimises the architecture for efficiency and modifies the model objective to predict the most likely word-context pairs [28]. As a result of this approach, words that appear in similar contexts throughout the training corpus end up embedded close to each other in vector space, which turns out to be beneficial in many applications [4].

Numerous methods have been developed to factor in the contextual information into the representation of words and word sequences. This thesis utilises the two popular approaches to learning distributed representations, namely Word2Vec [30] and Doc2Vec [31]. Both methods utilise a shallow neural network architecture with single hidden layer but use slightly different goals and strategies to learn semantic embeddings.

Word2Vec is designed to learn context-aware embeddings for individual words in one of two ways [30]: the *CBOW* model using context sequence to predict target words and the *skip-gram* model using a word to predict context. This thesis experimented with both models but employs only the latter because it performed better in the exploratory trials. The skip-gram model estimates the probability of a particular fixed-length context around the target word w. During training, the network learns a set of weights U from the input vector to the linear embedding layer h as well as a set of weights V from h to the output. After training, the rows of the projection matrix U hold the embeddings for individual words of the corpus vocabulary.

As a result of the learning procedure, Word2Vec produces a lookup table that maps the corpus vocabulary tokens to the corresponding embedding vectors of dimension $\dim(h)$. To form the representation of a log instance consisting of many tokens, this thesis sums all vectors corresponding to the tokens comprising that log and scales it by the log length, which is a popular technique in application engineering practice [4].

Doc2Vec extends the Word2Vec concept to learn embeddings for the entire sequence (doc) along with individual words by treating *docs* as additional vocabulary entries [24]. The original Doc2Vec paper has proposed two approaches to the task [31]: the *PV-DM* model learning a *doc* embedding alongside the target word embedding with respect to input context (similar to *CBOW* of Word2Vec) and the *DBOW* model learning the *doc* embedding directly using the objective to predict fixed-length text fragments randomly sampled from the input sequence (similar to *Skip-gram*). This thesis employs the *PV-DM* model, as it demonstrated better results in exploratory trials. The training approach is similar to that described for Word2Vec, except that in *PV-DM* model, the input vector contains an extra doc identifier, which provides an additional set of weights U' connected to the embedding layer h. After training, the rows of the matrix U' hold the numerical representations for every document in the corpus.

Notably, in the literature dedicated to software logs, multiple log-oriented semantic embedding models have been proposed. E.g., Log2Vec [39] is a prominent novel approach to log encoding that attempts to incorporate log-specific semantic features into embeddings as well as to tackle the *out-of-vocabulary word* problem during online encoding. Although Log2Vec offers many benefits for the software log classification task, it requires substantial prior knowledge of the application domain as well as considerable effort in incorporating this knowledge into the algorithm. Therefore, this approach would be better suited for an in-house team project rather than an independent scope-limited master's thesis work.

Stage 3 diagnostics

Because the encoding procedure is a preparatory step for the classification task, Stage 3 output can be roughly assessed by estimating the clustering quality of the resulting numerical representation of the dataset. In the ideal case, the points of the same class cluster together in vector space while maintaining a reasonable margin from the points of other classes. Hence, it is possible to think of any clustering method as a classifier [24]. Likewise, any classifier can thus be regarded as a supervised clustering method.

Based on the reasoning above, the classification algorithm trained at Stage 4 is the ultimate diagnostic tool for Stage 3 output. However, training an expressive classifier is a computationally expensive and time-consuming procedure. Hence, due to a large number of possible encoding variants in Stage 3, it would be helpful to filter out in advance those variants that result in the least favourable clustering properties and are poised to underperform in the classification task. For such a preliminary evaluation, this thesis employs the *k-nearest neighbour* (KNN) instance-based classifier method to assess the clustering properties of the encoded datasets produced by Stage 3 of the NLP pipeline.

The KNN is a supervised method that uses labelled points in the dataset as a reference for classifying new examples [24]. The most frequent class among the k nearest points in the training set determines the class of any point in the vector space. For the proximity metric, this thesis selected the Euclidean distance. Although generally, *cosine similarity* is considered a preferable similarity measure for text [62], the Euclidean distance becomes just as meaningful similarity measure when the encoded samples are normalised to the unit norm [24].

Unlike more sophisticated classification methods, KNN is fast to train and even faster to test. Furthermore, many popular classification algorithms can be seen as an extension of KNN [24], hence it is a good baseline to determine whether the dataset encoding would work well for other classifiers. The only downside is KNN's memory cost, as the classifier instance stores in memory the Voronoi regions across the whole dataset to facilitate quick prediction.

Initially, 20 different encoding schemes were considered for this thesis. However, based on the preliminary assessment using KNN, only seven encoding schemes described earlier in this section (see also Figure 13) were selected for the experiments.

Stage 3 implementation

The *text encoding* routine was implemented as a Python module, using the decision diagram in Figure 13 as a specification.

The count-based vector space representations as well as their LSI-compressed variants were implemented using Scikit-learn [68] library. Training of semantic embeddings has been implemented using the Gensim [72] software package.

Stage 4. ML model training and evaluation

The previous stage of the NLP pipeline has produced multiple variants of the dataset using various encoding schemes for its vector space representation. The purpose of Stage 4 is to train a potent classifier for each of these dataset variants and select one most likely to perform well in the field.

Learning a good predictor is an intricate problem. The primary challenge is to estimate the capacity of the ML model to generalise its predictions beyond training examples to any sample drawn from the same distribution [8]. Section 2.3 has presented the bias-complexity trade-off as a credible approach to addressing this challenge. In this thesis, this approach motivates the choice of methods used in Stage 4 design.

Algorithm 1 n-fold cross-validation

```
1:  input:
2:     ML classifier model embodying a hypothesis class $H$.
3:     A collection of hyper-parameter sets $\Theta \ni \theta$ for the model.
4:     Algorithm $A$ learning a predictor $h \in H$.
5:     Training set split into $n$ fragments: $S_T = S_1 \cup S_2 \cup ... \cup S_n$
6:  for each $\theta \in \Theta$
7:     for $i = 1...n$
8:        $h_{i,\theta} = A(S_T \backslash S_i; \theta)$        // note, $S_i$ is a validation fragment
9:     $e_{CV}(\theta) = \frac{1}{n}\sum_{i=1}^{n} L_{S_i}(h_{i,\theta})$   // cross-validation error for $\theta$
10: output:
11:    $\theta^* = \arg\min_\theta\{e_{CV}(\theta)\}$     // optimal hyper-parameter set
12:    $h_{\theta^*} = A(S_T; \theta^*)$             // optimal predictor object
```

Numerous strategies have been developed to learn a predictor model striking a good balance in terms of the bias-complexity trade-off. For theoretical research, *Shalev-Shwartz and Ben-David* [8] advice the *structural risk minimisation* (SRM) method for it employs a scientifically sound basis. However, this method imposes strong assumptions on the hypothesis class of an ML model and, even then, finding an optimal solution is an NP-hard problem, which makes SRM intractable in many cases [20].

Alternatively, the approach, known as *n-fold cross-validation*, is popular in application engineering practice for its simplicity and robustness. Even though [8] warn about the pitfalls related to the lack of rigorous base for this method, a

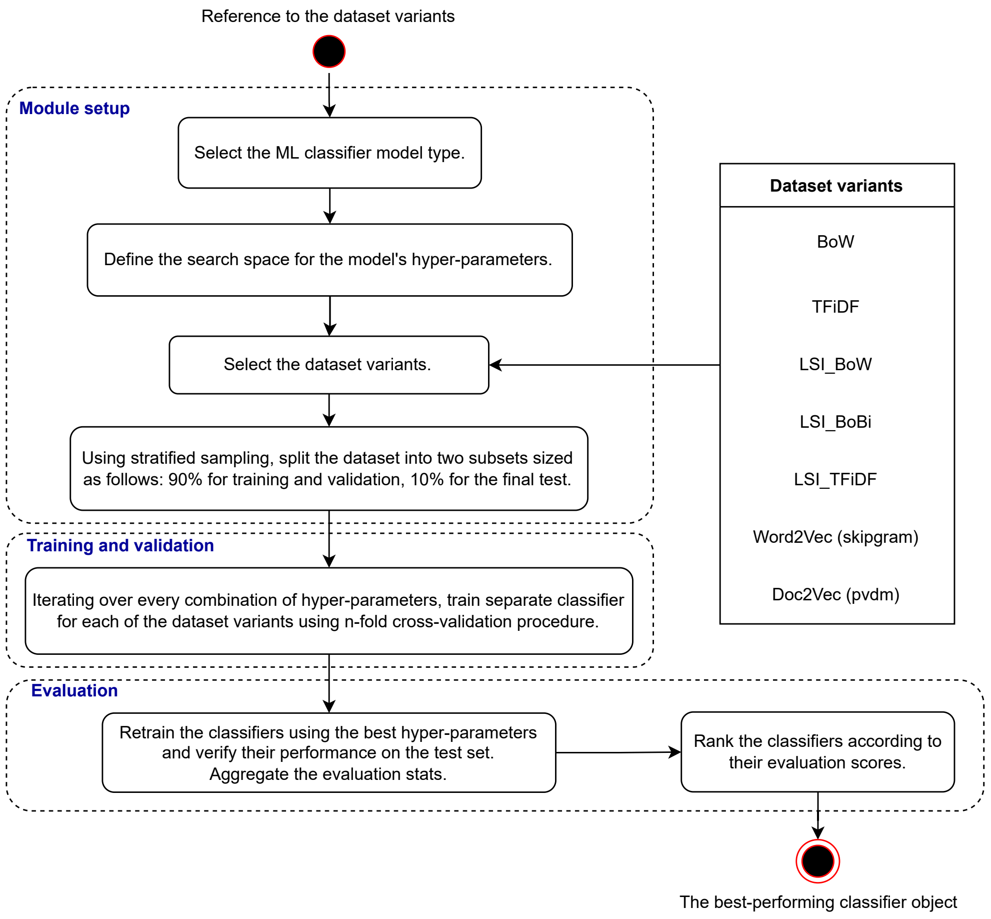

Reference to the dataset variants
Module setup
Select the ML classifier model type.
Define the search space for the model's hyper-parameters.
Select the dataset variants.
Using stratified sampling, split the dataset into two subsets sized as follows: 90% for training and validation, 10% for the final test.
Training and validation
Iterating over every combination of hyper-parameters, train separate classifier for each of the dataset variants using n-fold cross-validation procedure.
Evaluation
Retrain the classifiers using the best hyper-parameters and verify their performance on the test set. Aggregate the evaluation stats.
Rank the classifiers according to their evaluation scores.
The best-performing classifier object
Dataset variants
BoW
TFiDF
LSI_BoW
LSI_BoBi
LSI_TFiDF
Word2Vec (skipgram)
Doc2Vec (pvdm)

Figure 14: The decision diagram illustrating Stage 4 of the NLP pipeline. The module produces a collection of candidate classifier objects for each of the selected dataset variants, ranks them and selects the best-performing one.

more recent publication by *Mohri et al.* [20] provides evidence of learning guarantee for cross-validation with the upper bound for generalisation loss close to that of SRM. Therefore, this thesis employs *n-fold cross-validation* method as a systematic approach to learning a classifier.

The pseudocode of Algorithm 1 adopted from [8] presents the n-fold cross-validation procedure, which motivates the design structure and the flow of the *training and evaluation* stage of the NLP pipeline developed for this thesis. Figure 14 illustrates the main steps of this stage.

Module setup

First, one defines the type of ML classifier model. Based on the selected classifier type, one should specify its hyper-parameter settings θ. Typically, the optimal settings are not known in advance. Hence, a range of prospective values is defined for each of the model's arguments, thus forming the hyper-parameters' search domain Θ. This module has been designed so that any type of classifier implementing a compatible interface can be used for the task.

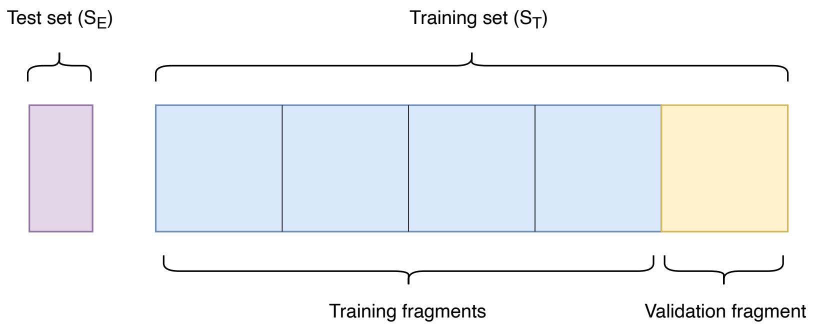


Figure 15: Dataset partitioning for training, validation and evaluation steps. The larger subset (the Training set S_T) is used for training and validation procedure, while the smaller one (the Test set S_E) is withheld for the final evaluation.

After defining the model type, one selects the dataset variants for the experiments. In preparation for training, validation and evaluation, each dataset variant is split using *stratified sampling* to ensure that all label classes are distributed proportionally between the subsets [8]. As depicted on Figure 15, a larger portion of the dataset S_T (90% in this thesis) is used for the *training and validation* step, while the remaining part S_E (10%) is reserved for the evaluation step.

Training and validation

In the training and validation step, a classifier object is instantiated. The module trains and evaluates the classifier, using *stratified n-fold cross-validation* procedure for every possible combination of hyper-parameter settings in the search domain. The goal is to find such settings that yield the best classifier performance. Such an exhaustive approach to finding the optimal hyper-parameter set is known as *grid search* [62].

Figure 16 provides a schematic representation of the training and validation routine developed for this thesis. The training set is split into $n = 5$ fragments using stratified sampling, where $n - 1 = 4$ fragments are applied for training and the remaining one for validation. The classifier is trained on the training fragments using the ERM rule (2), explained in Section 2.3. The validation fragment is applied to evaluate the predictive performance of the trained classifier. For each hyper-parameter set, the routine repeats n times so that every fragment serves for validation exactly once [8].

As a result, for m datasets and p hyper-parameter combinations, the 5-fold cross-validation procedure yields m tables each filled with p training and validation scores computed as average across 5 cross-validation rounds (1 per fold). In each such table, the results are ranked based only on the validation score because it is representative of the classifier performance in the field, whereas the average training score serves as a proxy for the expressive power of the model relative to the data at hand [20].

The choice of $n = 5$ folds for the cross-validation procedure has been motivated by two factors: the time to perform a cross-validation round and the variance in the

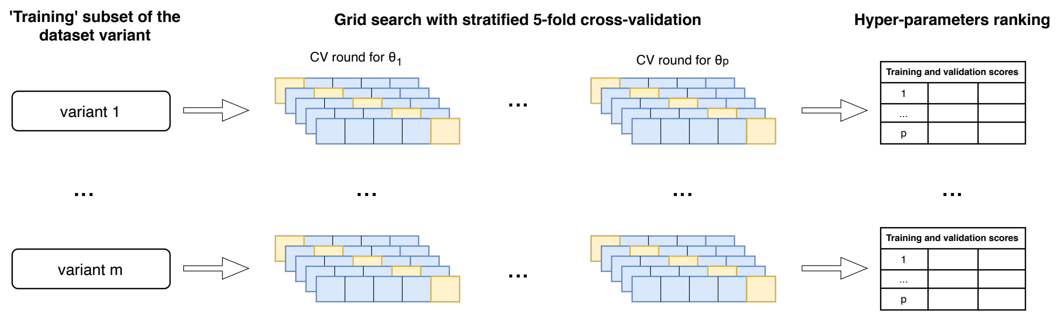

'Training' subset of the dataset variant
Grid search with stratified 5-fold cross-validation
Hyper-parameters ranking
CV round for θ1
CV round for θp
variant 1
...
variant m
Training and validation scores
1
...
p

Figure 16: Training and validation step employing stratified 5-fold cross-validation procedure performed on the selected classifier type for each possible combination of the hyper-parameter settings $\theta \in \Theta$.

validation scores. For a greater number of folds, the time to train a classifier increases, which leaves less time for other experiments. Additionally, a greater number of folds results in a smaller size of the validation data fragment, causing a greater spread in validation scores. Notably, $n = 5$ is the smallest number of folds recommended in the literature [20]. Although $n = 5$ happened to be the optimal choice for this study, the solution setup does not prevent from using a different number of folds.

Evaluation

In the final step of Stage 4, the classifier is retrained anew, this time, on the whole training set S_T, using the best hyper-parameters θ^* found during the training and validation step. The resulting model is then evaluated on the test set S_E, specifically withheld for this purpose. The final classifier ranking is based on the combination of validation and testing scores. Various aspects of results evaluation and ranking are discussed in more detail in Section 5.

Support vector machines (SVM)

Although the designed solution is not restricted in terms of model type, the scope of this work does not permit performing in-depth evaluation of more than one classifier for the baseline solution. One empirical study [73] has revealed that *random forest* (RF) and *support vector machines* (SVM) are the two prominent classifier types that are likely to achieve the best performance result on an arbitrary dataset. In this thesis, the kernelised version of SVM has been selected for the task. Compared to other potent options (e.g., RF), SVM permits more flexible fine-tuning of its expressive power via the selection of kernel type and the availability of the hyper-parameters for regularisation [20]. Such a control over the hypothesis class complexity is particularly desirable when analysing the classifier performance with respect to the bias-complexity trade-off.

SVM belongs to a family of linear classifiers [74]. Therefore, the hypothesis class H of an SVM is a set of labelling functions $h : \mathbb{R}^d \mapsto Y$ that employ separating hyperplanes $w^T x + w_0 = 0$ to discriminate between the classes $y \in Y$. In binary settings, when $Y = \{1, -1\}$, SVM predictor learns the optimal separator by maximising the *geometric margin* $\gamma(x) = y(w^T x + w_0)/\|w\|$ for all training examples $(x, y) \in S_T$, whereas the data points that end up closest to the hyperplane are referred to as *support vectors*.

In multiclass settings, when $|Y| = k > 2$, there exist various strategies to select a particular set of separators, such as one-versus-rest (OVR), one-versus-one (OVO) as well as direct methods. For SVM, the selected multiclass learning strategy determines the particular formulation of the learning problem [74]. Although the direct learning of k separating hyperplanes all at once receives much attention in the literature, an objective comparison between the multiclass learning strategies for SVM, done in [75], has found that OVO approach is among the preferable ones for practical use. Therefore, this thesis employs the OVO strategy for multiclass SVM implementation.

In OVO approach, the training data S_T is split into $\binom{k}{2} = k(k-1)/2$ subsets $S_{c,c'} \subset S_T$ containing the data points for each possible pair of classes c, c', $c \neq c'$. Within each pair, the class labels are reassigned so that label $c = -1$ and label $c' = +1$. The algorithm learns $\binom{k}{2}$ binary classifiers $h_{c,c'} : X \mapsto \{-1, 1\}$, whose decisions represent votes in favour of one of the two classes in each pair. Predictor outputs the class that attains the majority of votes in its favour. A tie can be resolved by randomly choosing one of the tied contenders.

For each of $\binom{k}{2}$ binary SVM classifiers $h_{c,c'}$, finding the optimal separator involves solving an optimisation problem, which can be formulated as follows [20]:

$$\begin{aligned} \min_{w, w_0, \xi} \quad & \frac{1}{2}\|w\|^2 + C_r \sum_{i=1}^{|S_{c,c'}|} \xi_i \\ \text{s.t.} \quad & y_i(w^T x_i + w_0) \geq 1 - \xi_i, \forall i \in \{1, ..., |S_{c,c'}|\} \\ & \xi_i \geq 0, \forall i \in \{1, ..., |S_{c,c'}|\}, \end{aligned} \tag{17}$$

where ξ_i are the slack variables facilitating the mechanism of *soft margin* and Cr is the regularisation parameter.

The problem (17) consists of quadratic objective function and a set of affine inequality constraints. This observation implies that the optimisation problem is convex and hence admits the unique optimal solution [76]. In addition, *Mohri et al.* [20] show that this setup satisfies the Karush-Kuhn-Tucker conditions, suggesting that *strong duality* holds at the optimal point. Reformulating the problem (17) using Lagrangian relaxation yields its dual form [20]:

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^{|S_{c,c'}|}} \quad & \sum_{i=1}^{|S_{c,c'}|} \alpha_i - \frac{1}{2} \sum_{i=1}^{|S_{c,c'}|} \sum_{j=1}^{|S_{c,c'}|} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \\ & \sum_{i=1}^{|S_{c,c'}|} \alpha_i y_i = 0, \quad \forall i \in \{1, ..., |S_{c,c'}|\} \\ & 0 \leq \alpha_i \leq C_r, \quad \forall i \in \{1, ..., |S_{c,c'}|\}, \end{aligned} \tag{18}$$

where α_i is a dual variable for each i-th example in the dataset $S_{c,c'}$, and the term

$K(x_i, x_j)$ is a kernel function efficiently performing the inner product operation in a high-dimensional space.

Once the solution for α is obtained, the predictor output can be computed as follows [20]:

$$h_{c,c'}(x) = sgn(\sum_{i=1}^{|S_{c,c'}|} \alpha_i y_i K(x_i, x) + w_0), \tag{19}$$

where $sgn : \mathbb{R} \mapsto \{-1, 1\}$ is the sign function, which returns -1 if the input is negative, 1 otherwise, and w_0 is obtained by computing $w_0 = y_v - \sum_{i=1}^{|S_{c,c'}|} \alpha_i y_i K(x_i, x_v)$ from any valid support vector x_v, for which $0 < \alpha_v < C_r$.

After $\binom{k}{2}$ classifiers $h_{c,c'}$ are trained, their votes are counted for each of the observations $x \in X$. Formally, the prediction $\hat{y}(x) \in C$ that yields the majority vote can be obtained as follows [20]:

$$\hat{y}(x) = \arg\max_{c' \in C} |\{c : h_{c,c'}(x) = 1\}| \tag{20}$$

Although there are many possible variations of the SVM optimisation problem setup [74], the dual representation (18) is especially convenient as it lends itself to the direct application of a variety of kernel functions [20]. This quality permits kernelised SVM learning also non-linear separators, thus making it a truly versatile classifier.

Stage 4 diagnostics

Thus far, little has been said about the metric used to evaluate a trained classifier. This is because there exist a variety of metrics to assess classifier performance, and either of them can be selected for training and evaluation.

Concerning each class $c \in C$ in the multinomial setting, all classifier predictions can be characterised in terms of four basic outcomes: either *true positives* (tp_c) or *true negatives* (tn_c) for correct predictions and either *false positives* (fp_c) or *false negatives* (fn_c) when predictions are incorrect. Various performance metrics can be defined using only these four quantities.

In this thesis, the metric used for training and evaluation scores is *accuracy* computed across all classes. It measures the fraction of correct predictions relative to all N predictions and is computed as follows:

$$Accuracy = \frac{1}{N} \sum_{c \in C} tp_c. \tag{21}$$

Accuracy, as defined in (21), treats all classes equally and hence favours a classifier that yields the maximum of correct predictions, no matter how unbalanced these predictions might be across classes [24]. The choice of accuracy as the primary metric for evaluation and ranking in this thesis is motivated by the application requirements. If the requirements change, the solution permits customising the ranking metric as well as specifying the class importance weights that are applied to the classifier objective function and hence affect prediction results.

In addition, the $precision_c = tp_c/(tp_c + fp_c)$ and $recall_c = tp_c/(tp_c + fn_c)$ metrics computed during the evaluation provide insight into the predictor's ability to distinguish a particular class c among others [77]. A special metric that combines these two into one is known as $F1_c = 2(precision_c \times recall_c/(precision_c + recall_c)$ [24]. This thesis, employs the *macro F1* measure to estimate the average F1-metric across all classes, defined as follows:

$$macro\ F1 = \frac{1}{|C|} \sum_{c \in C} F1_c. \tag{22}$$

As an alternative to *macro F1*, the Matthews Correlation Coefficient (MCC) has become also popular in the ML community [77]. It is deemed to provide more balanced measure of multinomial predictor performance compared to both F1 and accuracy. The MCC is defined as follows [77]:

$$MCC = \frac{N \sum_{c \in C} tp_c - \sum_{c \in C} (tp_c + fp_c)(tp_c + fn_c)}{\sqrt{(N^2 - \sum_{c \in C} (tp_c + fp_c)^2)(N^2 - \sum_{c \in C} (tp_c + fn_c)^2)}} \tag{23}$$

Stage 4 implementation

The *ML model training and evaluation* routine was been implemented as a Python module using the decision diagram Figure 14 as a specification. Classifier training was implemented with Scikit-learn [68] software library, including stratified splitting of the dataset, cross-validation, and the grid search routine that facilitates a series of of cross-validation rounds with various hyper-parameter settings.

For the kernelised SVM model, Scikit-learn provides a wrapper for the SVC class from the LibSVM library [78], which is considered one of the best open-source implementations of SVM [73]. In addition, the implementation utilises Intel's extension for Scikit-learn [79] patch that optimises the performance of many Scikit-learn models, including SVC, when training is executed on a compatible Intel processor.

4.3 Transfer Learning Solution

The previous section presented the baseline solution implementing a canonical NLP classification pipeline that relies on extensive data analysis and text preprocessing. This section describes an alternative approach that employs a deep neural network to learn high-quality text representation features from unprocessed text sequences. The most prominent of these learning machines have become banks of general knowledge applicable to a wide range of application domains [52]. The strategy, known as *transfer learning* (TL), permits adapting these pre-trained neural networks for domain-specific tasks [54], thus achieving competitive performance results while obviating the need for laborious preprocessing.

The transfer learning solution has been implemented using *Python* programming language. This chapter includes references to the software libraries used for the implementation.

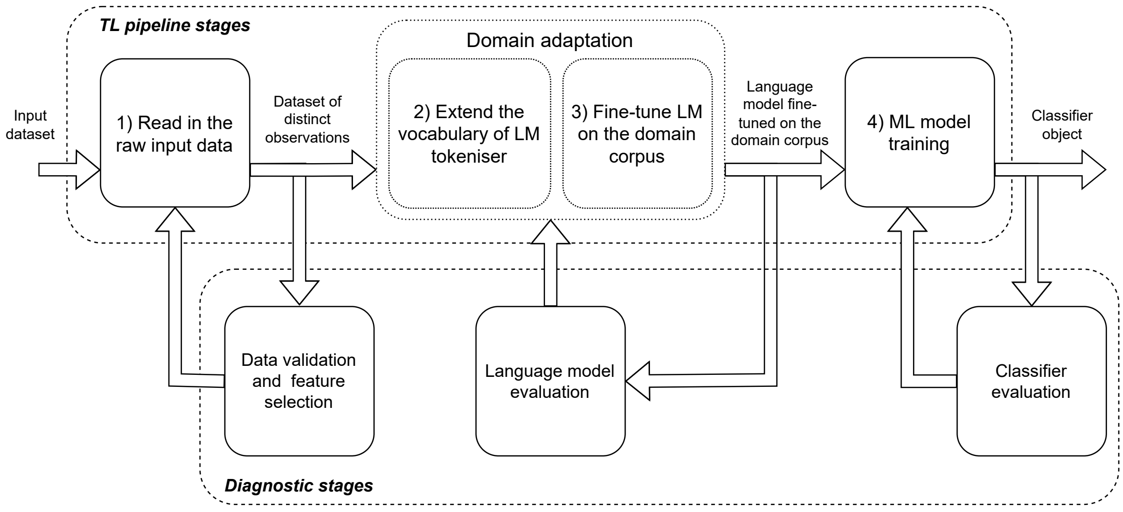

TL pipeline stages
Input dataset
1) Read in the raw input data
Dataset of distinct observations
Domain adaptation
2) Extend the vocabulary of LM tokeniser
3) Fine-tune LM on the domain corpus
Language model fine-tuned on the domain corpus
4) ML model training
Classifier object
Data validation and feature selection
Language model evaluation
Classifier evaluation
Diagnostic stages

Figure 17: The software pipeline implementing transfer learning approach with fine-tuning for a domain-specific content. The TL pipeline stages perform domain adaptation of a pre-trained language model for the classification task. The diagnostic stages provide insight into the output of the TL pipeline stages.

Transfer learning pipeline

Section 3.2 introduced the concepts of transfer learning and domain adaptation. This section presents the practical application of these concepts for solving the problem subject of this thesis. The schematic representation of the transfer learning solution is shown in Figure 17.

As seen from the figure, the TL solution is a software pipeline consisting of two main parts: the *TL pipeline stages* and the *Diagnostic stages*. The TL pipeline stages implement the domain adaptation procedure using the transfer learning approach that involves *intermediate fine-tuning* of a pre-trained language model (LM) on the domain-specific corpus and adapting the fine-tuned LM for the classification task. The diagnostic stages facilitate the evaluation of the TL pipeline stages results.

Notice that Stages 1 and 4 of the TL pipeline (along with their respective diagnostic stages) are identical to those of the NLP pipeline stages from Section 4.2. That is, the procedure for constructing a proper dataset from the raw input data as well as the classifier training routine remain the same as described in Section 4.2. The distinct feature of the transfer learning solution is the absence of both the text preprocessing and data encoding stages. Instead, the TL pipeline stages 2 and 3 implement the compound *domain adaptation* module that uses the dataset of distinct observations and the unprocessed text data to adapt a neural language model to the application domain. The diagnostic counterpart of the domain adaptation stage implements the language model evaluation routine.

Domain adaptation

In NLP, neural language models (NLMs) have served both as the mechanism for knowledge encoding as well as the medium for knowledge transfer [10]. Therefore, the

TL pipeline structure depicted in Figure 17 was designed such that it enables both pre-training and adaptation for the downstream task. The corresponding features are implemented within the *domain adaptation* module that consists of two distinct stages. Stage 2 provides the means to set up the corpus vocabulary and train the language model tokeniser. Stage 3 facilitates the language modelling task in order for the model to capture the domain-specific knowledge.

Such a setup enables a flexible approach to domain adaptation. For example, in the abundance of training data from the application domain, the TL solution permits configuring and pre-training a language model from scratch, as it has been done for SciBERT [60]. Although such an approach is the most favourable in terms of performance expectations, neither the dataset nor the computational resources available for this study make it feasible to pre-train an NLM from scratch. Hence, this study places the main focus on utilising the prior knowledge of a pre-trained NLM for domain adaption.

When using a pre-trained NLM, there are three main approaches for domain adaptation with knowledge transfer [54]. In the *feature-based* approach, the NLM is used to generate numerical representations of the input data similar to the semantic embeddings produced in Stage 3 of the NLP pipeline. In the *fine-tuning* approach, the NLM encoder module is trained on the domain-specific data, thus further improving numerical representation with respect to both the application domain and the downstream task. In the *prompt-based tuning* approach, an elaborate training method is used specifically to narrow the gap between the source and the target domains to improve the quality of knowledge representation.

The TL pipeline design does not restrict the choice of approach for knowledge transfer. However, the scope of this work provides the opportunity to experiment with only one of the three methods described above. Based on the literature review, the fine-tuning approach appears to be the most popular as well as the most effective for classification [55], [56], [60]. Therefore, this thesis employs the fine-tuning method for domain adaptation with a pre-trained NLM.

Fine-tuning for classification

The transfer learning practice with NLM has established multiple fine-tuning strategies suited for different circumstances and application setups. *Kalyan et al.* [54] have determined four main fine-tuning strategies that branch further into eight different sub-strategies. Due to the scope constraints, this section only describes the mainstream strategies selected for this study for their proven track record in classification, namely *vanilla* and *intermediate* fine-tuning with the focus on domain adaptation.

Vanilla fine-tuning for classification employs a pre-trained NLM encoder with the classification output layer on top. The model is trained for several epochs on domain-specific data, during which all neural network layers participate in learning. Since the resulting model is an actual classifier, the training is performed using the cross-validation routine precisely as it is implemented in Stage 4 of the NLP pipeline described in Section 4.2. In this thesis, vanilla fine-tuning is the default adaptation strategy facilitated automatically, as it requires nothing more than instantiating an

NLM with the classification head and passing it directly to Stage 4 for training.

The intermediate fine-tuning strategy extends the vanilla principle into a multi-step adaptation process, in which the NLM can be trained for other related tasks or using additional datasets from a relative domain (or both). In this thesis, the intermediate fine-tuning is implemented as a two-step process. In the first step, the NLM vocabulary is extended to accommodate the terms of the target domain. The model is then trained in the masked language modelling task using the dataset of the target domain. In the second step, the model encoder is refitted for classification and is then passed on to Stage 4 to continue its domain adaptation, following the procedure described for the vanilla fine-tuning strategy.

Masked language modelling (MLM) task

The selection of the masked language modelling objective for step 1 of the intermediate fine-tuning strategy is motivated by the need to accommodate the context information specific to the application domain. This is traditionally achieved by exercising language modelling on the domain-specific corpus [10]. The MLM training procedure utilises the whole input sequence, a small part of which is hidden (masked) at random. The model learns to predict the hidden sequence elements by using only the context formed by the visible parts around the hidden ones. This training approach enables the model to utilise context from either side of the hidden elements, thus accommodating bi-directional information in the resulting numerical representations [54].

Although *Kalyan et al.* [54] point out several shortcomings of MLM compared to several alternative language modelling methods, MLM remains the mainstream approach commonly available in open-source language models. Besides, many of the state-of-the-art transformer models have been pre-trained using MLM [55], [56]. Hence, MLM is a reliable choice, although is not fixed and can be switched to an alternative option upon availability.

Domain adaptation diagnostics

Before the domain adaptation procedure, the target dataset is split into training and test sets in the 9:1 proportion. Such an arrangement permits evaluating the first adaptation step when using the intermediate fine-tuning strategy to ensure consistent adaptation results. The same test set used to assess the MLM results is later used for the classifier evaluation. Hence, the test set is never used for training neither in step 1 nor in step 2 of the adaptation routine.

Just like any other language modelling task, MLM is evaluated using perplexity measure, formally defined as follows [28]:

$$2^{-\frac{1}{n}\sum_{i=1}^{n}\log_2 LM(w_i|w_{1:i-1})}, \tag{24}$$

where $LM(w_i \mid w_{1:i-1})$ is the probability of the word w_i given its preceding context $w_{1:i-1}$, predicted by the language model LM.

The perplexity scores are computed for the NLM fine-tuned on the domain-specific corpus for both training and test sets to ensure the consistency of results.

Neural language model selection

As explained in Section 3.2, the attention-based transformer has become the dominant model architecture in NLP and has been particularly successful in transfer learning [54]. Therefore, this study considers only transformer language models for the experiments. The TL solution in this thesis is designed to use the *transformers* API by Hugging Face [51], thus opening a vast selection of transformer models from the library as well as from the Hugging Face community. However, given the limited scope of this study, only one model type can be considered for the experiments.

Because this thesis relies on knowledge transfer, an important criterion would be the domain of the corpus used to pre-train a candidate model [46]. Ideally, the source model should be pre-trained on a corpus related to the software development or software testing domain. However, during this study, no public model was found to be pre-trained from scratch on such a specialised corpus. Therefore, this study narrowed the choice of language models to those pre-trained on general corpora.

Another important consideration is the maximum sequence length the model can process. The (tokenised) input sequence length of a typical transformer relying on the full attention mechanism is limited to 512 tokens. However, in the dataset used in this study, about 15% of logs are longer than 512 tokens in length, not to mention that the subword tokenisation approach used in many modern transformer modifications tends to split words into fragments, thus further increasing the actual sequence length. The tokens that do not fit into the limit are cropped out, which carries a risk of losing essential information. For this reason, the choice of the candidate models was further reduced to those that admit input sequences of at least 2000 tokens in length. This requirement preserves the entire length of $\approx 99\%$ of observations in the used dataset.

Among the general-purpose models that admit sufficiently long input sequences, two prominent candidates were identified: the BigBird [80] model from the BERT-family of transformers and the GPT3 [59] model from the GPT family. However, at the time of writing this thesis, GPT3 has not been open for public access. Therefore, this thesis selected the BigBird transformer model for the TL solution experiments.

BigBird transformer

BigBird has come into prominence for its *sparse attention* mechanism, which gains 8-fold improvement in computational efficiency compared to the *full attention*. The creators of BigBird have managed to show that the costly *full attention* mechanism can be retired in favour of a more lightweight approach that scales at lesser computational cost, permits longer sequences without losing the most important properties of the original transformer, and even achieves the new state-of-the-art performance in several NLP tasks [80].

BigBird has been pre-trained on general corpora using MLM task, starting from a public checkpoint of RoBERTa [56]. Same as RoBERTa, BigBird utilises the subword byte-pair-encoding tokenisation method and incorporates bi-directional context information in its text encodings.

Implementation

The solution has been implemented using a large version of pre-trained BigBird model, made available via Huggingface [51] transformers NLP library as *google/bigbird-roberta-large*. Training such models even in the domain adaptation mode requires potent and expensive hardware, particularly high-end GPUs. The Google Colab Pro+ service has been used to run the transfer learning solution implementation to make the domain adaptation experiments possible.

Although conceptually, Stage 4 of the transfer learning pipeline is identical to that of the baseline solution, its implementation could not be directly reused due to the specifics of the neural networks training procedure and because the Pytorch [81] model class of the BigBird interface is different from that of Scikit-learn models used in the baseline solution. Therefore, Stage 4 has been implemented anew along with the domain adaptation stages 2-3 in the Colab environment.

In addition, the *Simple Transformers* [82] wrapper package for the Huggingface library has been used as a reference for implementing domain adaptation and transformer classifier. The *Weights & Biases* [83] ML platform has been used for experiment tracking.

5 Results

Section 4 presented two learning-based solutions developed for this thesis. This chapter describes the experiments conducted to evaluate these solutions on the dataset provided by the client as well as presents and interprets the results of these experiments.

This chapter is organised as follows. Section 5.1 evaluates the baseline solution at each stage of the NLP pipeline. Section 5.2 evaluates the transfer learning solution with the primary focus on the domain adaptation and classifier evaluation stages.

5.1 Baseline Solution Evaluation

This section presents the experimental results obtained from the baseline solution described in Section 4.2. The experiments involve passing the dataset through the NLP pipeline stages and collecting the information about the output from the respective diagnostic modules.

Stage 1 - constructing the dataset

In the first pipeline stage, the raw input data is organised into a dataset of distinct observations consisting of features applicable for learning. Accomplishing this task requires proper insight into the composition of the input data. This block presents the insights gained from the Stage 1 diagnostic module and explains the decisions concerning the dataset content.

The data collected for this study spans a period of 7 months, accruing a total of 105407 failed test report instances. Figure 18 presents the dynamics of raw data accumulation each week during this period. The figure also reflects the distribution of failed tests across different software *targets*, thus highlighting the problem areas in the software development environment.

As shown in the figure, thousands of failed test reports accrue per week. However, the majority of these are duplicates produced from repeated test runs. Of the 105407 data points in the dataset, only 29318 are distinct (*Text Data*, *Label*) pairs. Figure 19 summarises the composition of raw data in terms of its duplicate content and presents the distribution of distinct data points across target variants. For this thesis, the distinct data points are of particular interest because they represent unique examples required in order for the ML algorithms to learn.

Failure category labels

Section 4.1 introduced the in-house report analyser tool used by the client for data labelling. Creators of the tool have defined 75 unique *category* labels corresponding to the suspected causes of failure. The dataset contains observations for 68 unique failure categories presented in Figure 20.

Evidently, the distribution of failure classes is very unbalanced. The first 10 most frequent labels account for 90% of all distinct data points. Moreover, many classes accrued only a handful of observations. In order to obtain a representative summary

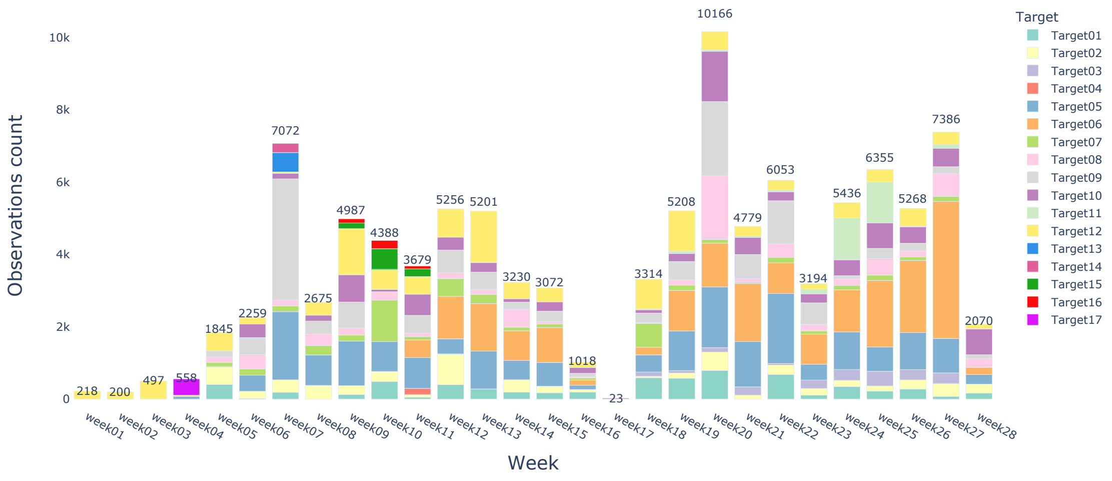
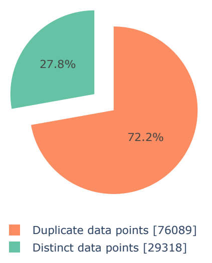
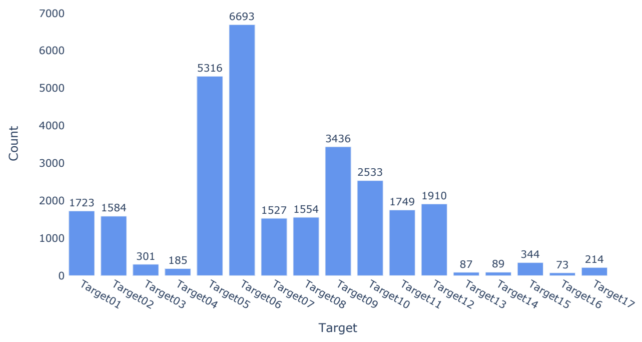


Figure 18: The timeline showing the number of data points (failed test reports) acquired per week from different software target variants. It spans the time period of 28 weeks, accruing 105407 data points in total.



(a) Raw data composition



(b) Distinct data points across software target variants

Figure 19: Composition of the dataset used in this thesis. The pie chart in Figure 19(a) illustrates the share of distinct data points. The bar chart in Figure 19(b) illustrates the distribution of distinct data points across target variants.

of the classifier performance, there should be a sufficient number of examples for training, validation, and testing. Therefore, in the interest of this study, the client organisation has agreed to exclude the failure categories that accrued less than 30 observations.

In addition, the client has identified 8 *special classes*, which involved sophisticated data preprocessing by the report analyser system. Because the provided dataset does not include the information from these preprocessing steps, the special classes are also excluded from the scope of the learning task.

Finally, 4 parent categories are used to help in organising the multitude of classes into logical groups. The composition of each parent category is presented in Figure 21.

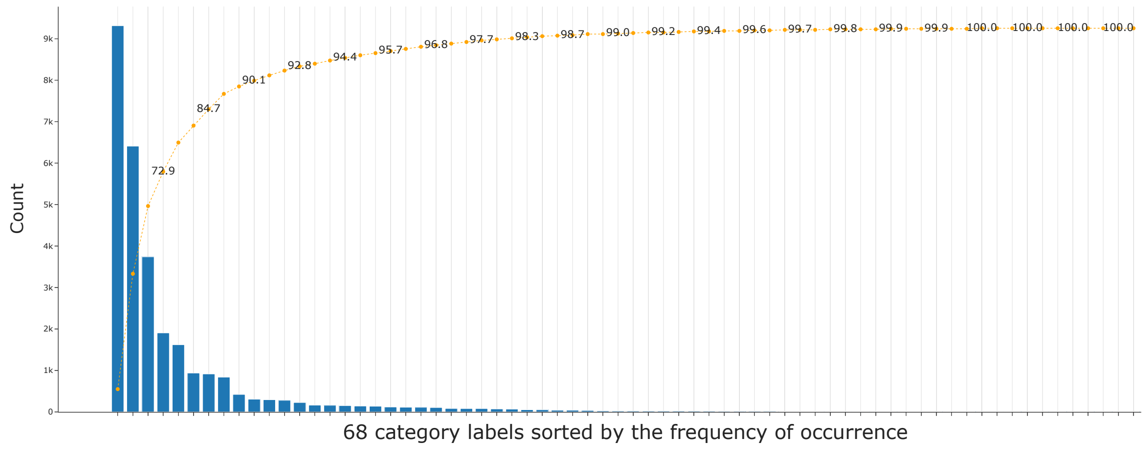


Figure 20: Distribution of failure category labels sorted by the frequency of occurrence in the dataset. The cumulative sum plot displays the percentage of data points accumulated along with label observations.

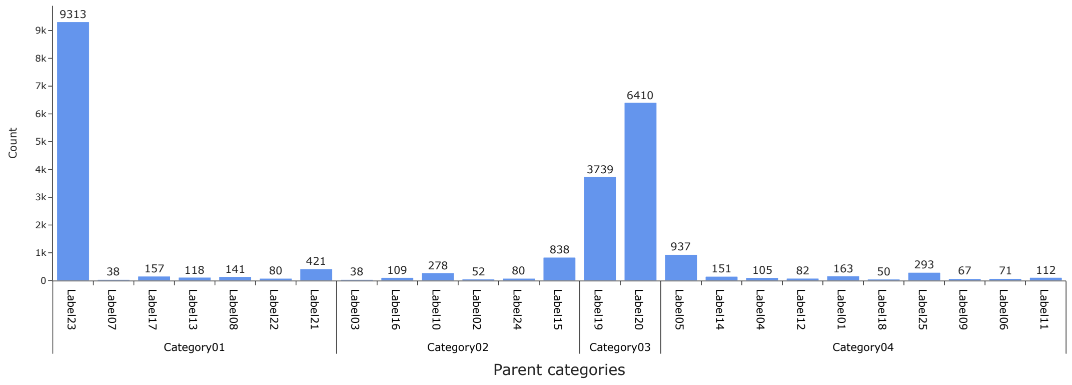
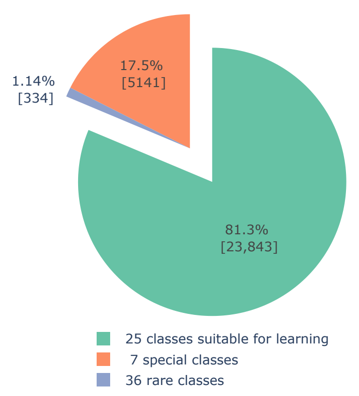


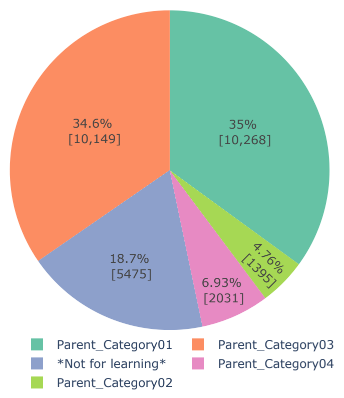
Figure 21: Number of observations for the class labels selected for the learning task. The labels are grouped by the respective parent categories.

The figure illustrates only the classes selected for the learning task.

Figure 22 presents the dataset partitioned in terms of different label groups. As a result of this partitioning, 25 failure category labels have been selected for the classifier learning task, accounting for 81.3% of all distinct observations in the dataset. While this is a considerable reduction in the scope of classification task, the client has agreed that this is an acceptable compromise, given the data available. Table 1 summarises the composition of the dataset before and after partitioning into label groups.



(a) Dataset without rare and special classes



(b) Parent categories

Figure 22: Dataset composition in terms of label groups. Figure 22(a) presents the share of classes selected for the learning task. Figure 22(b) presents the dataset partitioned in terms of 4 parent categories and 1 additional super-class composed of the special and the rare classes excluded from learning.

Table 1: Dataset composition in terms of label groups. Each entry is presented in $(n, [c])$ format, where n is the number of distinct observations and c is the number of classes under the given parent category.

	Not for learning		Parent category				
	Special class	Rare class	Category01	Category02	Category03	Category04	Total
Before partitioning	0, [0]	0, [0]	11839, [25]	1476, [13]	11904, [4]	4099, [26]	29318 , [68]
After partitioning	5141, [8]	334, [36]	10268, [7]	1395, [6]	10149, [2]	2031 [10]	23843 , [25]

It is important to emphasise that the number of classes selected for the learning task is not a limitation of the learning-based approach, but rather a design parameter freely adjustable upon need. As more observations accumulate over time, the pipeline can be reconfigured to include more classes.

Feature selection

In addition to text data and labels, test reports contain metadata entries, such as the earlier mentioned *Target* identifier, which also characterise the dataset and may be of value for a classification algorithm. Notably, all these variables, except for the *Date* and *Text data*, happen to be purely categorical. The great number of modalities makes it impractical to study them with the naked eye. Instead, these variables are tested for meaningful associations with the output using the χ^2 test.

Table 2 summarises the results of the χ^2 test of independence performed for the categorical input variables and *Label* output variable. For all selected input variables, the p-value approaches 0, indicating a low risk of rejecting the null hypothesis of independence. These results were further validated with the help of bi-variate correspondence analysis. Therefore, all these variables were considered prospective learning features.

Table 2: Results of the χ^2 tests to assess independence between each of the categorical input variables and the output *Label* variable. For all tests, the *p-value* approaches 0, indicating a low risk of rejecting the null hypothesis.

χ^2 test for *Label* with:	Target	Test Setup	Test Group	Test Data	Build Info Key
significance alpha	0.01	0.01	0.01	0.01	0.01
degrees of freedom	1072	2680	3350	248168	12998
chi-square critical	1182.65	2853.25	3543.36	249809.88	13376.02
chi-square score	22662.90	44498.30	38997.00	471763.73	94269.53
total inertia	0.77	1.52	1.33	16.09	3.22
p-value	$\approx 10^{-12}$	$\approx 10^{-14}$	$\approx 10^{-19}$	$\approx 10^{-17}$	$\approx 10^{-19}$

Table 3 summarises the composition of the dataset free of duplicate data points and redundant variables. The table presents the number of unique modalities for each of the variables. This data serves as the input for the next NLP pipeline stage.

Table 3: Structure of the dataset obtained as a result of data transformation in the 1st NLP pipeline stage. The table summarises the data content in terms of number of variable modalities.

			Input features X						Output labels Y	
	index	Date	Build info key	Target	Test setup	Test group	Test data	Text data	Label	Parent
number of modalities	23843	154	191	17	41	51	3643	23637	25	4

Stage 2 - text data preprocessing results

Text preprocessing has involved considerable experimentation effort. This subsection summarises the raw text data content at the input of Stage 2 and the most prospective results of the preprocessing procedure.

Text data analysis

While metadata in the failed test reports provides some clues about a probable cause of the issue, the most precise information is contained in text data, as it includes error messages and stack traces generated exactly at the moment of failure. For this reason, this solution relies primarily on log text data for classification. Figure 23 gives a glimpse of the most typical log contents.

In the context of this study, the collection of all text logs in the dataset are referred to as a *log corpus*. This corpus size is equal to the number of distinct data points in the dataset, that is 29318 log instances. The collection of all unique terms that occur in the corpus as independent lexical units, such as words, prepositions,

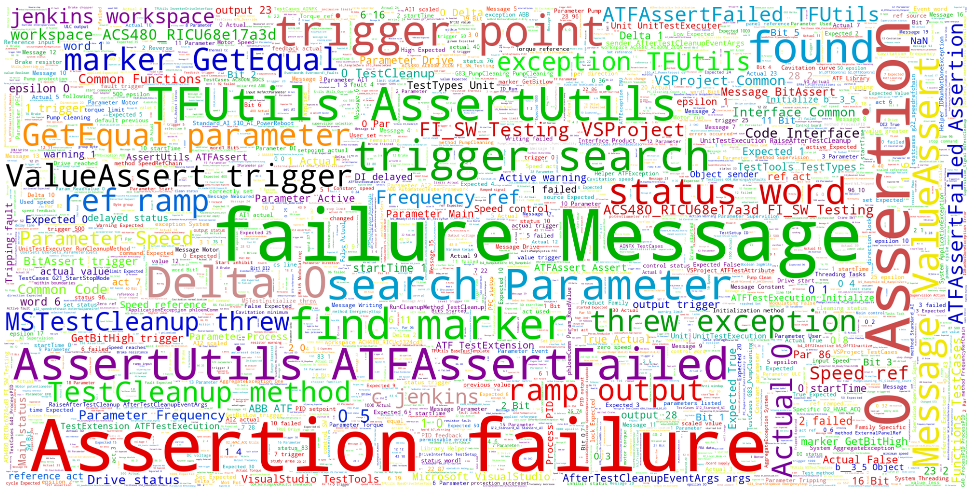


Figure 23: Cloud of token collocations most frequently occurring in the failed test logs. Text of larger size implies higher frequency of occurrence in the log corpus. Diverse text colouring aids readability.

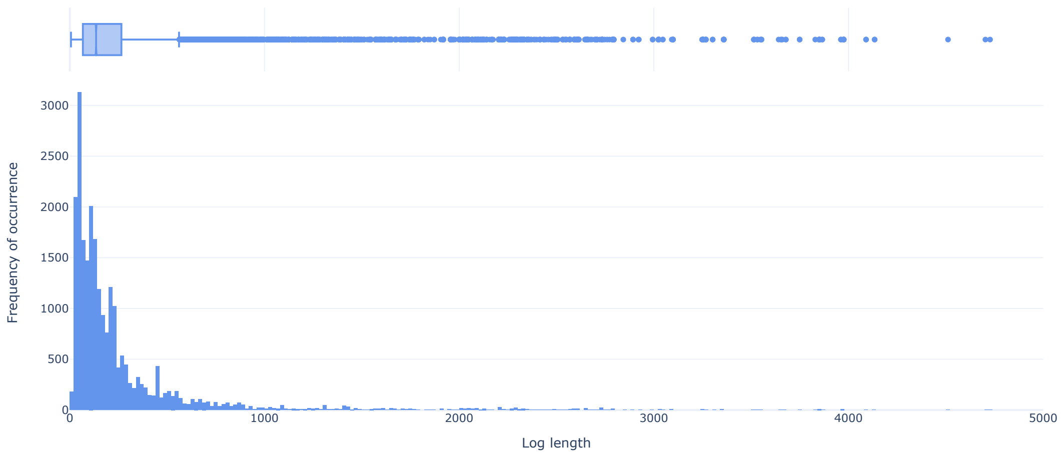


Figure 24: Log length distribution across the log corpus. The marginal box plot outlines the robust measures for centre and spread of the distribution.

articles, and standalone numbers form the *corpus vocabulary*. The vocabulary of this log corpus contains 77617 unique tokens, 67955 of which are numerals of some form.

Another measure that characterises the corpus composition is the length of individual logs. Figure 24 gives an overview of the log length distribution across the log corpus. The lengths appear to be binomially distributed with 3/4 of all logs not exceeding 265 tokens in length. There is a prominent positive skew with a long right tail which extends to an observation containing 14141 tokens, whereas the shortest log instance is only 6 tokens long. The marginal box plot whiskers span an extra 3/2 of interquartile range (IQR), reaching the 90th percentile of observations at the length of 562 tokens. A more detailed quantitative summary can be found in Table 4.

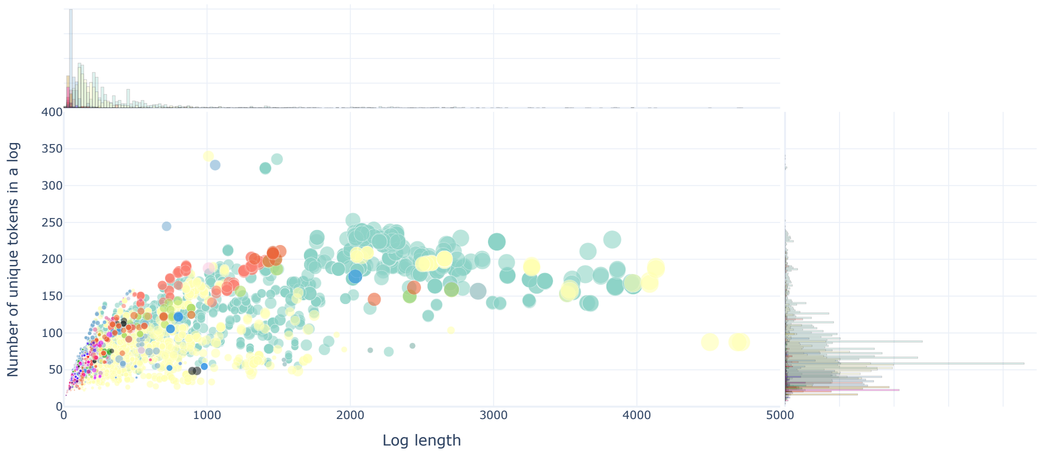


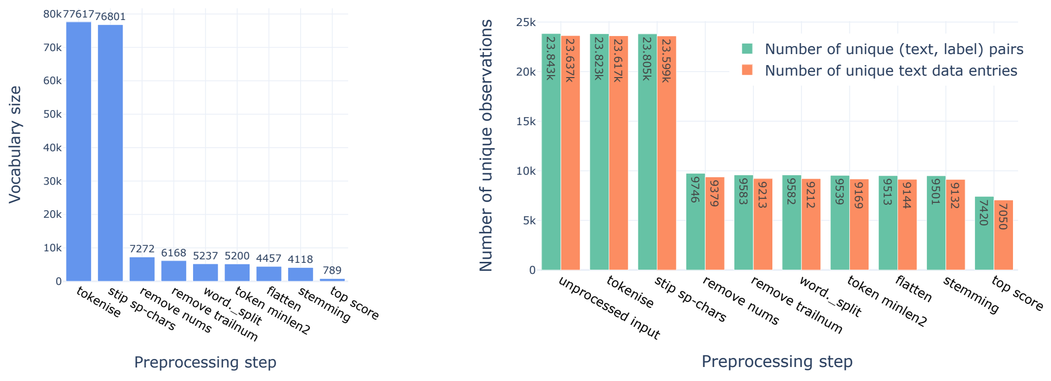
Figure 25: Number of unique tokens in a log versus its length. Each circle represents one log message and its radius is proportional to the number of numerals in its text. Different colours correspond to different failure categories. The marginal histograms outline the distribution mass of the axes variables.

Although the logs in this corpus vary significantly in length, even the longest instances consist of only a small fraction of the corpus vocabulary. Figure 25 illustrates the trend in the number of unique tokens occurring in logs of different lengths. Notably, even extremely long log instances spanning thousands of words rarely use more than 250 unique tokens, whereas the vast majority use less than 200 unique tokens. Such a low concentration of unique tokens gives reason to suspect a considerable amount of repeated text fragments in longer logs.

Table 4: Statistical figures that characterise this corpus in terms of log length and numeral-to-length count ratio. The values are provided for complete logs and for the same logs but composed of unique tokens only.

	Log length					Numeral-to-length ratio		
	range	median	IQR	mean	std	IQR	mean	std
Complete logs	[6, 14141]	135	[67, 265]	266	464.68	[0.08, 0.14]	0.1	0.05
Unique tokens	[5, 340]	57	[39, 78]	63	36.31	[0.11, 0.24]	0.18	0.11

Table 4 quantitatively summarises the values for log length distribution and the share of numerals in these logs. This data suggests anomalies in length distribution and reveals a relatively high ratio of numerical tokens. These preliminary findings motivate the choice of preprocessing steps, the results of which are presented in the next block.



(a) Effect of preprocessing on the log corpus vocabulary size.

(b) Counts of distinct observations and unique log text instances at each preprocessing step.

Figure 26: Text preprocessing overview. Figure 26(a) presents the change in vocabulary size with each preprocessing step. Figure 26(b) presents the change in number of distinct observations and unique text samples in the corpus.

Preprocessing results

Figure 26 presents the effect of the preprocessing procedure on the composition of the dataset used in this thesis. As shown in the figure, the corpus vocabulary size has been reduced to 789 unique tokens, which is close to 1% of its original size (77617 tokens).

As seen in Figure 26(a), the most dramatic reduction in the vocabulary size occurs after removal of numerical tokens. This preprocessing step also turns about 60% of the data points into duplicates, which is manifested as a drop in distinct observations count in Figure 26(b). This effect suggests that a considerable number of logs are identical except for a few numerals. The analysis of these numerals using the *token ranking* method (described in Section 4.2) revealed that most of these do not help to discriminate between classes. The few that were found to have predictive potential, such as reoccurring error code numbers, were retained in the vocabulary.

Notably, as seen in Figure 26(b), the number of unique text logs is consistently lower than the number of distinct data points, which indicates that certain log instances have identical text but are labelled differently. This is an important observation that highlights the presence of a stochastic component in the data. This aspect will be discussed in more detail in Section 6.

The effect of preprocessing on the log length distribution is presented in Figure 27. The figure indicates a considerable reduction in spread of the preprocessed text data compared to that of the original corpus. A more detailed summary is given in Table 5. As seen from the table, the concentration of unique tokens per log has also decreased proportionally. This implies that, despite preprocessing, the contingency of repeated text in the log samples has remained roughly the same.

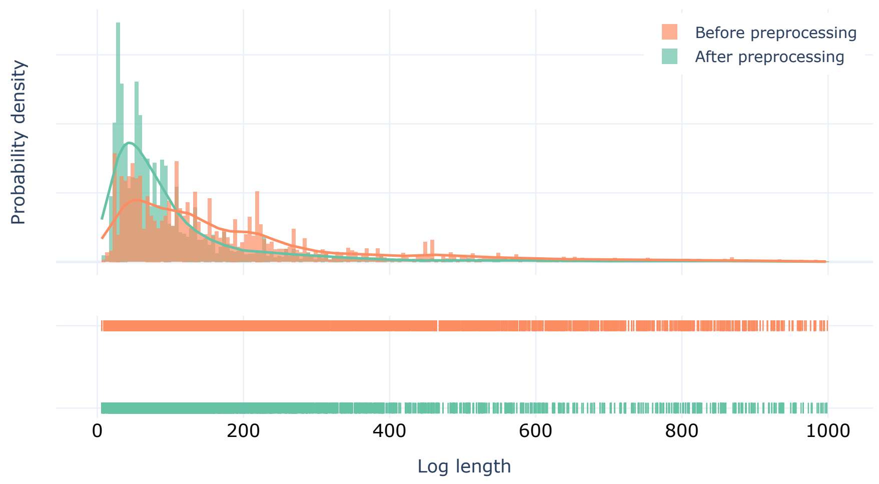


Figure 27: Log length distribution before and after preprocessing at Stage 2 of the NLP pipeline. The marginal rug plot illustrates the density of the observations found for different log lengths.

Table 5: Text data length before and after preprocessing. The values are provided for complete logs and for the unique tokens content only.

	Before preprocessing					After preprocessing				
	range	median	IQR	mean	STD	range	median	IQR	mean	STD
Complete logs	[6, 14141]	130	[63, 246]	248	459.55	[6, 5324]	61	[33, 131]	132	245.84
Unique tokens	[5, 340]	55	[39, 77]	62	34.83	[3, 115]	29	[22, 43]	34	16.29

Stage 3 - dataset encoding

In Stage 3, the dataset is converted into a high-dimensional vector space representation interpretable by a machine learning algorithm. The quality of representation might vary depending on the selected text encoding scheme. One of the goals of this study is to evaluate the effect of different encoding schemes on the classifier performance. In order to facilitate the experiments with different representations, Stage 3 produces multiple variants of the dataset using the encoding schemes presented in the description of Stage 3 in Section 4.2.

Table 6 summarises the output of Stage 3. In this case, all dataset encoding variants have the same number of observations and the same number of classes selected for the learning task at Stage 1. Although Stage 2 text preprocessing has revealed many of the raw data points to be nearly identical, these have been originally identified as distinct learning examples and hence should be preserved for the experiments.

Table 6: Different encoding scheme representations of the dataset summarised in terms of dataset composition and vector dimensions. The dimension values are provided for two configurations of the dataset: one consisting of text data features only and second including the metadata variables.

Encoding schemes:	Dataset composition # of observations	 # of classes	Text only log vector dimension	Metadata included log vector dimension
BoW	23843	25	789	794
TFiDF	23843	25	789	794
LSI_BoW	23843	25	100	105
LSI_BoBi	23843	25	500	505
LSI_TFiDF	23843	25	500	505
Word2Vec_SG	23843	25	200	205
Doc2Vec_pvdm	23843	25	200	205

In the table, each encoding scheme is characterised by a number of variables, corresponding to the dimension of a log instance vector space representation. Note that in addition to text data features, the dimension of the log representation vector might also include the metadata variables selected at Stage 1. To assess the predictive potential of these metadata variables, separate experiments are conducted with and without the use of metadata by the learning algorithm.

LSI variants of the dataset have been constructed for BoW, BoBi and TF-IDF vector space representations. The number of LSI features has been selected to preserve at least 99% of the original information. The semantic embeddings *Word2Vec* and *Doc2Vec* were trained using settings recommended by the Gensim [72] library documentation. The dimension of the vector space representation was set according to the log corpus vocabulary size and was also verified by empirical trials.

Stage 4 - SVM classifier evaluation results

In Stage 4, a machine learning model is trained and evaluated using the methods presented in the description of the NLP pipeline Stage 4 in Section 4.2. The experiments are conducted individually for every encoding scheme produced in Stage 3 in two dataset configurations (Table 6): the first using text data features only and the second with metadata variables included.

This thesis employs the kernelised version of SVM classifier for multinomial classification. The first set of experiments is conducted with the *linear* kernel SVM $K(x_i, x_j) = x_i^T x_j$ to establish the baseline performance using the simplest form of discriminator function (i.e., linear). An additional set of experiments is conducted with the *radial basis function* (RBF), also known as *Gaussian*, kernel $K(x_i, x_j) = e^{-\gamma\|x_i - x_j\|^2}$ to determine the effect of a more expressive hypothesis class on the predictor performance.

Every set of experiments utilises its own hyper-parameter search space. For the SVM implementation used in this work, the hyper-parameter interface is that of the SVC class from the LibSVM [78] library, wrapped as a Scikit-learn [68] estimator. Table 7 lists the hyper-parameter settings used for the classifier training. The most important hyper-parameter is the regularisation constant C that permits fine-tuning the classifier soft margin. For the RBF kernel variant, the parameter *gamma* (γ) plays an important role in defining the coverage of the decision boundary. Other hyper-parameters were set as per the problem setup or kept default.

Table 7: Hyper-parameter settings for the *sklearn.svm.SVC* classifier object used in the experiments. The entries in brackets [...] signify multiple possible options, which, in combination with others, form the hyper-parameter search space.

SVM hyper-parameters:	**Linear kernel**	**RBF kernel**
C (regularisation constant)	[range of values]	[range of values]
kernel	'linear'	'rbf'
gamma (γ)	-	['scale', 'auto']
shrinking	False	False
tol (stopping tolerance)	1e-4	1e-4
class_weight	None	None
decision_function_shape	'ovo'	'ovo'

The hyper-parameter selection is facilitated by the *grid search* routine, an integral part of the 5-fold cross-validation procedure used for classifier training and evaluation. The procedure requires the dataset to be split into fragments. Table 8 gives the breakdown of the number of observations after splitting. Stratified sampling ensures that each subset contains examples for each of the 25 classes selected for learning.

Table 8: Number of points and classes in each of the subsets after splitting.

Dataset splitting	Training set	Validation set	Test set
Number of observations	17167	4291	2385
Number of classes	25	25	25

The results of training, validation, and testing are summarised in the Tables 9 to 12. For the training and validation procedure, these tables aggregate the best average accuracy values μ_{train} and μ_{val} as well as the respective standard deviations σ_{train} and σ_{val} of the scores computed across 5-fold cross-validation runs. For the final evaluation, the classifier yields predictions for the test set data. These predictions are used to compute a range of evaluation metrics described in Section 4.2 for Stage 4 diagnostics.

Linear kernel SVM results

Table 9 and Table 10 present the results of the experiments conducted with linear kernel SVM. In both tables, *Word2Vec_SG* and *Doc2Vec_pvdm* encoding variants achieve the highest average validation score $\mu_{val} \approx 0.960$, though this result is not far ahead of those attained by *TFiDF* and *LSI_BoW*.

Table 9: Evaluation results for the linear kernel SVM classifier trained and tested on different encoding variants of the dataset using text data only. The results are ranked in descending order of μ_{val} score.

Text only	Training and validation				Evaluation on the test set				
dataset	μ_{train}	σ_{train}	$\mu_{\mathbf{val}}$	σ_{val}	acc	μ_{prec}	μ_{recall}	mcc	macro F1
Word2Vec_SG	0.9688	0.0017	**0.9622**	0.0016	**0.9656**	**0.9617**	**0.9656**	**0.9538**	**0.8514**
Doc2Vec_pvdm	0.9680	0.0009	0.9584	0.0018	0.9577	0.9561	0.9577	0.9432	0.7815
LSI_BoW	0.9675	0.0009	0.9562	0.0016	0.9560	0.9519	0.9560	0.9409	0.7730
TFiDF	0.9659	0.0008	0.9552	0.0018	0.9610	0.9599	0.9610	0.9476	0.7950
BoW	0.9680	0.0007	0.9546	0.0013	0.9551	0.9530	0.9551	0.9398	0.7740
LSI_TFiDF	0.9657	0.0009	0.9508	0.0020	0.9589	0.9578	0.9589	0.9448	0.7999
LSI_BoBi	**0.9710**	0.0005	0.9460	0.0013	0.9551	0.9520	0.9551	0.9399	0.7827

The evaluation on the test set demonstrates that the classifier accuracy matches or even surpasses the validation score attained by the cross-validation procedure. Both precision μ_{prec} and recall μ_{recall} scores appear balanced, suggesting no excessive occurrence of false positives nor false negatives.

Table 10: Evaluation results for the linear kernel SVM classifier trained and tested on different encoding variants of the dataset using text+metadata variables. The results are ranked in descending order of μ_{val} score.

Text+metadata	Training and validation				Evaluation on the test set				
dataset	μ_{train}	σ_{train}	$\mu_{\mathbf{val}}$	σ_{val}	acc	μ_{prec}	μ_{recall}	mcc	macro F1
Word2Vec_SG	0.9710	0.0019	**0.9621**	0.0020	**0.9669**	0.9638	**0.9669**	**0.9555**	**0.8600**
Doc2Vec_pvdm	0.9696	0.0010	0.9603	0.0016	0.9648	**0.9641**	0.9648	0.9528	0.8372
TFiDF	**0.9774**	0.0003	0.9594	0.0023	0.9648	0.9635	0.9648	0.9528	0.8085
BoW	0.9733	0.0010	0.9571	0.0011	0.9618	0.9603	0.9618	0.9488	0.8060
LSI_BoW	0.9692	0.0012	0.9570	0.0021	0.9610	0.9581	0.9610	0.9478	0.8085
LSI_TFiDF	0.9759	0.0006	0.9549	0.0018	0.9648	0.9637	0.9648	0.9527	0.8119
LSI_BoBi	0.9726	0.0004	0.9509	0.0021	0.9614	0.9590	0.9614	0.9484	0.8132

Comparing the results in the two tables demonstrates an insignificant effect of the metadata variables on the general performance of linear kernel SVM. The only notable difference concerns the *macro F1* score, which is higher in the evaluation results of the experiments using metadata information along with the text data variables. This difference implies that metadata might improve the accuracy of the underrepresented classes that are otherwise difficult to separate.

RBF kernel SVM results

Table 11 and Table 12 present the results of the experiments conducted with the RBF kernel SVM. Compared to the linear kernel results, the increase in classifier expressive power helps to improve the performance of the semantic-aware encoding variants but has a mixed effect on the others. As a result, *Word2Vec_SG* and *Doc2Vec_pvdm* variants achieve the best class separability across the board.

Table 11: Evaluation results for the RBF kernel SVM classifier trained and tested on different encoding variants of the dataset using text data only. The results are ranked in descending order of μ_{val} score.

Text only	Training and validation				Evaluation on the test set				
dataset	μ_{train}	σ_{train}	$\mu_{\mathbf{val}}$	σ_{val}	acc	μ_{prec}	μ_{recall}	mcc	macro F1
Word2Vec_SG	0.9724	0.0006	**0.9648**	0.0030	**0.9698**	**0.9698**	**0.9698**	**0.9595**	**0.8635**
Doc2Vec_pvdm	0.9763	0.0003	0.9642	0.0013	0.9677	0.9652	0.9677	0.9566	0.8291
LSI_BoW	**0.9780**	0.0004	0.9579	0.0016	0.9627	0.9593	0.9627	0.9499	0.7845
BoW	0.9764	0.0004	0.9490	0.0010	0.9497	0.9480	0.9497	0.9323	0.7358
LSI_BoBi	0.9770	0.0004	0.9487	0.0016	0.9560	0.9534	0.9560	0.9409	0.8051
TFiDF	0.9726	0.0006	0.9462	0.0022	0.9493	0.9472	0.9493	0.9318	0.7401
LSI_TFiDF	0.9713	0.0004	0.9304	0.0033	0.9342	0.9333	0.9342	0.9113	0.7006

Notably, in both tables, the lowest-ranking encoding variants exhibit a relatively large disparity between their training μ_{train} and validation μ_{val} scores. Such a disparity is a typical manifestation of the *overfitting* phenomenon explained in Section 2.3. In contrast, among the top-ranking variants, the gap between μ_{train} and μ_{val} scores is considerably lower.

Table 12: Evaluation results for the RBF kernel SVM classifier trained and tested on different encoding variants of the dataset using text+metadata variables. The results are ranked in descending order of μ_{val} score.

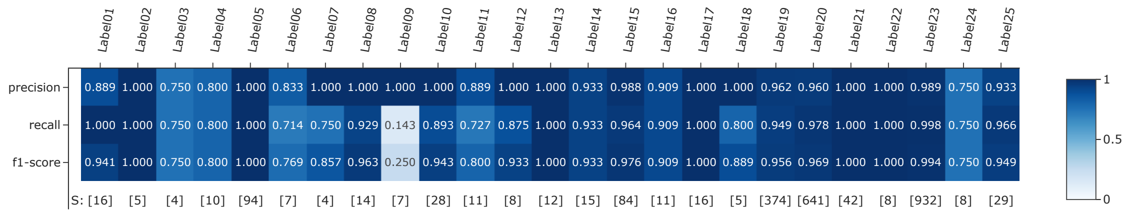
Text+metadata	Training and validation				Evaluation on the test set				
dataset	μ_{train}	σ_{train}	$\mu_{\mathbf{val}}$	σ_{val}	acc	μ_{prec}	μ_{recall}	mcc	macro F1
Word2Vec_SG	0.9773	0.0007	**0.9673**	0.0037	**0.9732**	**0.9733**	**0.9732**	**0.9640**	**0.8933**
Doc2Vec_pvdm	0.9786	0.0004	0.9652	0.0013	0.9690	0.9657	0.9690	0.9583	0.8300
LSI_BoW	0.9798	0.0004	0.9629	0.0022	0.9702	0.9676	0.9702	0.9601	0.8300
BoW	0.9811	0.0003	0.9522	0.0008	0.9551	0.9539	0.9551	0.9397	0.7739
LSI_BoBi	0.9846	0.0003	0.9522	0.0003	0.9572	0.9569	0.9572	0.9427	0.8186
TFiDF	**0.9850**	0.0005	0.9486	0.0025	0.9505	0.9510	0.9505	0.9337	0.7697
LSI_TFiDF	0.9819	0.0003	0.9341	0.0033	0.9392	0.9387	0.9392	0.9182	0.7295

Overall, the test set evaluation scores agree with those obtained by the cross-validation procedure. Similarly to the results of the experiments with the linear kernel, the use of metadata variables has a positive effect on the *macro F1* score of all encoding variants in the RBF kernel experiments.

Best SVM estimator performance summary

Based on the results presented above, the best performance scores are attained by the RBF kernel SVM classifier on the dataset variants utilising metadata variables. Therefore, the best baseline solution can be selected from Table 12.

In all experiments, the semantic-aware encoding variants have been observed to consistently outperform the other options, achieving the highest validation scores μ_{val} with a minor spread in results σ_{val}. Although the score difference between *Word2Vec_SG* and *Doc2Vec_pvdm* is marginal, the former is selected as the best performing encoding variant, as it also achieved the highest *macro F1* score.



	Label01	Label02	Label03	Label04	Label05	Label06	Label07	Label08	Label09	Label10	Label11	Label12	Label13	Label14	Label15	Label16	Label17	Label18	Label19	Label20	Label21	Label22	Label23	Label24	Label25
precision	0.889	1.000	0.750	0.800	1.000	0.833	1.000	1.000	1.000	1.000	0.889	1.000	1.000	0.933	0.988	0.909	1.000	1.000	0.962	0.960	1.000	1.000	0.989	0.750	0.933
recall	1.000	1.000	0.750	0.800	1.000	0.714	0.750	0.929	0.143	0.893	0.727	0.875	1.000	0.933	0.964	0.909	1.000	0.800	0.949	0.978	1.000	1.000	0.998	0.750	0.966
f1-score	0.941	1.000	0.750	0.800	1.000	0.769	0.857	0.963	0.250	0.943	0.800	0.933	1.000	0.933	0.976	0.909	1.000	0.889	0.956	0.969	1.000	1.000	0.994	0.750	0.949
S:	[16]	[5]	[4]	[10]	[94]	[7]	[4]	[14]	[7]	[28]	[11]	[8]	[12]	[15]	[84]	[11]	[16]	[5]	[374]	[641]	[42]	[8]	[932]	[8]	[29]

Figure 28: Multiclass report summarising the best performing SVM classifier predictions for each label in terms of precision, recall, and f1 score. The numbers below each column quantify the label-specific examples used in the evaluation.

Figure 28 provides a summary of RBF kernel SVM multiclass performance on the test set using *Word2Vec_SG* vector space representation. Despite a seemingly high general accuracy score $acc \approx 0.97$ (see Table 12), not every class enjoys such a high prediction accuracy. As seen from the figure, the observations of Label 9 are among the most likely to be misclassified, although its poor results have a minor impact on the final score due to a small proportion of examples representing this label in the test set support. Notably, some of the most frequently misclassified labels are those that happen to be the least represented, often having less than 10 examples in the test set. A more detailed insight into the classification results can be obtained from a confusion matrix in Figure A1.

5.2 Transfer Learning Solution Evaluation

This section presents the experimental results obtained from the transfer learning (TL) solution described in Section 4.3. The experiments involve passing the dataset through the TL pipeline stages and collecting information about the output from the respective diagnostic modules.

Preparing language model for domain adaptation

Stage 1 of the TL pipeline is identical to that of the baseline solution. Hence, the dataset composition remains the same as presented in Section 5.1 for the Stage 1 results, as summarised in Table 3. Stage 2 in the TL solution prepares the language model (LM) for training on the domain-specific corpus. The main challenge in the second stage is to set up the LM tokeniser to recognise tokens from a domain-specific vocabulary.

Generally, transformer tokenisers are able to automatically preprocess a set of documents to learn new tokens directly from the corpus. However, training a tokeniser turned out to be not straightforward in this case. For example, the analysis of text data in Section 5.1 showed that the raw log corpus of the given dataset contains an excessive number of arbitrary numerals, which exceeds the vocabulary size limit that a tokeniser could manage. Besides, the ability to recognise and encode numerical tokens is expected to be a core feature of the BERT-based language models [84].

Following this reasoning, two tokeniser configurations, shown in Table 13, were prepared for the domain adaptation stage. First, using the original LM vocabulary without changes. Second, extending the original vocabulary with domain-specific tokens, excluding numerals.

Table 13: The data and the two tokeniser vocabulary configurations prepared for the LM training: the *basic* variant using the original LM vocabulary and the *extended* variant adding the domain-specific vocabulary to the basic one.

	Data for LM training		LM vocabulary		
	Training set size	Test set size	Original model vocabulary size	Domain-specific vocabulary size	Tokens total
Basic	21458	2385	50358	0	50358
Extended	21458	2385	50358	6168	55581

As seen from the table, data for LM trainng is split into Training and Test subsets using the stratified random sampling method. In either configuration, the same unprocessed log text data is utilised. The only difference is the selected vocabulary content. Because of the overlap between the original and the domain-specific vocabularies, the total number of tokens in the composite vocabulary of the *Extended* tokenisers configuration does not equal the sum of comprising vocabulary sizes. In the following stages, the models with different vocabularies will be trained and tested to illustrate the effect of vocabulary choice on performance in the downstream task .

Language model fine-tuning

In Stage 3, a pre-trained neural language model is instantiated with the *masked language modelling* (MLM) head for fine-tuning on the domain-specific corpus. This study uses the large variant of RoBERTa-based BigBird model. The model is configured to use either Basic or Extended version of the BigBird tokeniser prepared in the previous stage (Table 13).

Generally, configuring a neural language model for training requires substantial expertise [51]. Therefore, this study relies on the preconfigured variant of the BigBird model, provided via the Huggingface library, and places the main focus on setting up the batch size parameters and the adaptive learning rate of the *AdamW* optimiser to achieve a consistent reduction in loss during the MLM training and evaluation process.

Figure 29 presents one of the typical MLM fine-tuning rounds carried out in this thesis. As seen in the figure, the most considerable reduction in loss occurs within the first five epochs of training. After that, the downward slope diminishes, thus yielding only marginal improvement in the performance metrics. Such behaviour is normal in neural network training practice [25], and it signals that the optimiser has likely converged close to a local minimum of the loss function.

The results of the MLM fine-tuning procedure for all BigBird configurations used in this study are presented in Table 14. The main difference between the fine-tuning rounds are the choice of vocabulary (Table 13) and the number of training epochs.

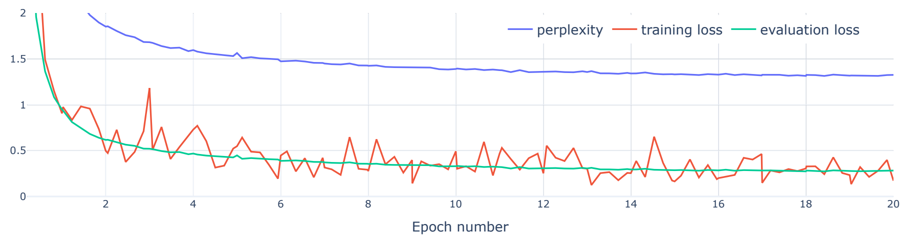


Figure 29: BigBird language model fine-tuning trend spanning 20 epochs of training in a masked language modelling task. Training loss gauges model performance on the Training set batches, whereas evaluation loss and perplexity scores reflect model performance on the Test set.

The performance metrics are presented for both training and test sets, whereas the former was used to learn software log representations and the latter was used to verify the language model performance on the previously unseen samples.

Table 14: Training and evaluation results for the BigBird model fine-tuned for MLM task in different configurations. The results are ranked in ascending order by evaluation loss.

BigBird large	Training set evaluation		Test set evaluation	
MLM results	training loss	perplexity	evaluation loss	perplexity
Basic ft20	0.1609	1.1745	0.1646	1.1789
Extended ft45	0.1898	1.2090	0.2349	1.2648
Extended ft20	0.2584	1.2949	0.2765	1.3185
Extended ft0	6.4034	603.9	6.45	631.92
Basic ft0	12.0215	166287.9	11.98	159390.53

As seen from the table, no adaptation at all (Basic ft0) predictably yields the worst MLM performance on the domain-specific corpus. Merely extending the vocabulary with domain-specific tokens (Extended ft0) already improves the results. Further fine-tuning (ft>0) on the training set drives the losses even lower until the optimiser saturates, as per the example in Figure 29. Remarkably, the model using the original vocabulary achieves the best results in MLM task. The effect of the fine-tuning on the downstream classification task are evaluated in the next stage.

BigBird classifier evaluation results

After the language model is fine-tuned on the corpus of software logs, its MLM head is replaced with the classification layer on top of the encoder module. In this configuration, the transformer model becomes a complete end-to-end classifier that is passed to Stage 4 for training and evaluation on the labelled data.

In the transfer learning solution, Stage 4 is no different from that of the baseline solution described in Section 4.2. The classifier undergoes the same cycle of cross-validation procedure to determine the best set of hyper-parameters and gauge its generalisation capacity. In classifier configuration, most of the default settings were retained, while the main effort was placed on tuning the optimiser and finding a good span for the training procedure.

Table 15: Evaluation results for the BigBird classifier fine-tuned on the log corpus using two different tokeniser configurations. The results are ranked in descending order of μ_{val} score.

Text only dataset	Training and validation				Evaluation on the test set				
	μ_{train}	σ_{train}	$\mu_{\mathbf{val}}$	σ_{val}	acc	μ_{prec}	μ_{recall}	mcc	macro F1
Extended ft45	**0.9697**	0.0003	**0.9605**	0.0017	0.9631	0.9596	0.9631	0.9504	**0.8574**
Extended ft20	0.9685	0.0008	0.9603	0.0022	**0.9656**	**0.9620**	**0.9656**	**0.9538**	0.8461
Extended ft0	0.9623	0.0007	0.9558	0.0017	0.9623	0.9576	0.9623	0.9493	0.8306
Basic ft20	0.9602	0.0012	0.9480	0.0040	0.9526	0.9493	0.9526	0.9365	0.8211
Basic ft0	0.9557	0.0010	0.9449	0.0046	0.9476	0.9428	0.9476	0.9296	0.7904

The results of BigBird classifier training and evaluation are presented in Table 15. The first observation is that the ranking based on classification performance does not match that of the MLM task obtained in Stage 3 (Table 14). That is, the low perplexity of a base language model does not necessarily translate into a superior classification accuracy. An important finding here is that extending the vocabulary alone without fine-tuning (Extended ft0) already makes the language model a better classifier than its non-extended version fine-tuned to excel in the MLM task (Basic ft20).

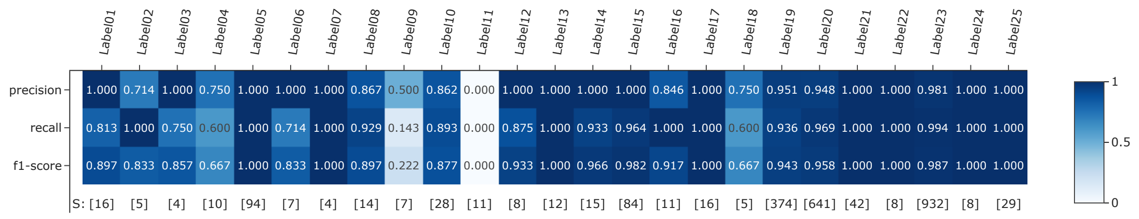
As seen from the table, fine-tuning the model with extended vocabulary improves the classification scores, although the gains seem marginal. Based on the μ_{val} and *acc* scores, fine-tuning adds up to ≈ 0.005 points in accuracy. On the one hand, longer fine-tuning (Extended ft45) yields little improvement compared to a moderate stride (Extended ft20), on the other, it does seem to improve the *macro F1* score.

Overall, both training and validation scores are stable, exhibiting very minor variation σ_{train} and σ_{val} during cross-validation. The evaluation on the test set revealed no anomalies, since *acc* agrees with μ_{val} score as well as μ_{prec} and μ_{recall} appear balanced, which is also reflected in *mcc* score.

Best BigBird estimator performance summary

Based on the evaluation results shown in Table 15, the best performance in terms of μ_{val} score was achieved by the model with extended vocabulary fine-tuned for a longer period of time (Extended ft45). Figure 30 illustrates the classifier performance results for each label individually.

As seen from the figure, more than half of the labels achieve nearly perfect scores both in terms of precision and recall. However, some labels, such as Label04 and Label18, achieved mixed results, while Label09 and Label11 were almost entirely confused with other labels. Just like in the case of the baseline solution, the poorly performing labels are heavily underrepresented in the dataset. A more detailed insight into the classification results can be obtained from a confusion matrix in Figure A2.



	Label01	Label02	Label03	Label04	Label05	Label06	Label07	Label08	Label09	Label10	Label11	Label12	Label13	Label14	Label15	Label16	Label17	Label18	Label19	Label20	Label21	Label22	Label23	Label24	Label25
precision	1.000	0.714	1.000	0.750	1.000	1.000	1.000	0.867	0.500	0.862	0.000	1.000	1.000	1.000	1.000	0.846	1.000	0.750	0.951	0.948	1.000	1.000	0.981	1.000	1.000
recall	0.813	1.000	0.750	0.600	1.000	0.714	1.000	0.929	0.143	0.893	0.000	0.875	1.000	0.933	0.964	1.000	1.000	0.600	0.936	0.969	1.000	1.000	0.994	1.000	1.000
f1-score	0.897	0.833	0.857	0.667	1.000	0.833	1.000	0.897	0.222	0.877	0.000	0.933	1.000	0.966	0.982	0.917	1.000	0.667	0.943	0.958	1.000	1.000	0.987	1.000	1.000
S:	[16]	[5]	[4]	[10]	[94]	[7]	[4]	[14]	[7]	[28]	[11]	[8]	[12]	[15]	[84]	[11]	[16]	[5]	[374]	[641]	[42]	[8]	[932]	[8]	[29]

Figure 30: Multiclass report summarising the best performing BigBird classifier predictions for each label in terms of precision, recall, and f1 score. The numbers below each column quantify the label-specific examples used in the evaluation.

While the lack of balance in the dataset is a strong factor explaining the classifier bias towards labels with a greater number of observations, there are also labels, such as Label22 and Label24, that achieve perfect classification scores despite being in a substantial minority. Clearly, there are also other reasons for misclassification, which will be discussed in detail in Section 6.

6 Discussion

The previous chapter presented the results of data analysis and the outcomes of the experiments with two candidate solutions. This chapter compares the competing solutions in terms of standard classification metrics and discusses the key findings obtained from the experiments as well as the implications of these findings. The chapter also provides a deeper analysis of the obtained classification results and attempts to assess reasonable classification performance expectations with respect to the theory and the specifics of the given dataset.

6.1 Comparison of the Solution Candidates

To facilitate comparison between the candidate solutions, Table 16 aggregates the results of the best performing classifiers evaluated in the previous chapter. For the baseline solution, the table presents the RBF kernel SVM classifier results in two dataset configurations: SVM (Text only) and SVM (Text+metadata), taken from Table 11 and Table 12 respectively. For the transfer learning solution, the table also contains the evaluation results of the BigBird (Text only) transformer model with extended vocabulary fine-tuned in the masked language modelling task (Table 15).

Table 16: Summary of the best classification results obtained from the experiments with the candidate solutions. The results are ranked in descending order of μ_{val} score.

	Training and validation				Evaluation on the test set				
	μ_{train}	σ_{train}	$\mu_{\mathbf{val}}$	σ_{val}	acc	μ_{prec}	μ_{recall}	mcc	macro F1
SVM RBF (Text+metadata)	0.9773	0.0007	**0.9673**	0.0037	**0.9732**	0.9733	0.9732	0.9640	**0.8933**
SVM RBF (Text only)	0.9724	0.0006	**0.9648**	0.0030	**0.9698**	0.9698	0.9698	0.9595	**0.8635**
BigBird (Text only)	0.9697	0.0003	0.9605	0.0017	0.9631	0.9596	0.9631	0.9504	0.8574

Based on the ranking in the table and the results of the *independent two-sample t-test*, the baseline solution appears to outperform the transfer learning solution by a narrow margin. Compared to BigBird, the SVM text-only solution variant achieves a higher score in every metric, and the comparison of the μ_{val} scores yields a *p-value* of ≈ 0.025, suggesting that the score difference is significant in this case. The text+metadata variant achieves even higher scores, although its advantage over the text-only SVM solution variant in terms of μ_{val} score is not statistically significant (*p-value* ≈ 0.278) due to the relatively high spread σ_{val}.

However, the performance scores alone do not explain the reasons for the obtained results. Therefore, it is also important to examine the factors that make these particular solutions stand out from other options. One of the crucial factors determining classification performance is the quality of data representation in vector space [24]. To address this concern, this thesis has dedicated much of the development effort to text preprocessing in Stage 2 of the baseline solution, which helped to gain insight into the characteristics of the vast corpus vocabulary and to reduce this vocabulary to a manageable size by removing noisy content. This effort permitted constructing count-based vector space representations composed of the features with the highest discriminative potential. However, when examining the classification performance (Tables 9 to 12), the best scores were obtained with the semantic embeddings *Word2Vec* and *Doc2Vec*, which required no preprocessing at all. In other words, an automatic contextual encoding procedure turned out to be more effective than the encoding that relied on careful token selection facilitated through preprocessing.

Apparently, in the process of preparing data for count-based representations, preprocessing inadvertently destroys information when reducing vocabulary size and removing the noise component from the text. In contrast, the context-aware representations can capture meaningful associations and encode them in a compact vector form even if the data is noisy. Another advantage of the semantic embeddings is the ability to extract information encapsulated in the context itself [28]. For example, a rare word that is likely to be removed during preprocessing can become a powerful discriminative feature in a context-aware encoding scheme through the mechanism of distributed representation, which maps semantically related words close to each other in vector space [30], [31].

If context-aware representation is indeed the key to superior performance in the log classification task, then the best possible solution should be the one most adept at extracting context information from data. Compared to static semantic embeddings, such as *Word2Vec* and *Doc2Vec*, the attention-based neural language models have been shown to be more potent in learning contextual embeddings for words [32] as well as for word sequences [33]. Based on this premise, the BigBird attention-based transformer model used in the transfer learning solution has the potential to learn more effective vector space representations for logs and to outperform the baseline solution in the log classification task. Yet, the empirical results presented in Table 16 do not agree with this proposition.

One possible explanation for the underwhelming performance of the transfer learning solution could be that achieving better quality contextual representations of software logs using transformer language models requires a specialised approach different from other application domains. For example, the experiments with BigBird fine-tuning revealed that vocabulary management is crucial for effectively adapting to the domain of software logs (see Table 14). This finding can be explained by the fact that software logs utilise a vast number of custom tokens that are unlikely to be found in the vocabularies of general-purpose language models. Another aspect that makes the domain of software logs special is the prevalence of stack traces of long compound tokens composed of shorter dot-separated terms. Commonly, these compound tokens are formed by permutations of the same short terms, thus artificially generating new vocabulary entries for a language model. This peculiar aspect creates the risk of combinatorial problems for vocabulary management and might become a source of continuous performance issues.

To mitigate this risk, the input text data might require preprocessing in order to split the compound tokens into shorter constituents, as is done in the baseline solution. Although it is rather unconventional to preprocess the input of a language model, an additional experiment with the BigBird model using both preprocessed data and the domain-specific vocabulary (similar to those constructed for the baseline solution) demonstrated a noticeable improvement in performance compared to that using unprocessed data, as shown in Table 6.2. Word splitting increases vocabulary overlap (and hence similarity) between the source and the target model domains, which is a likely explanation for the observed performance improvement.

Table 17: Performance results of the BigBird model fine-tuned and trained for classification on the preprocessed text data using the reduced domain-specific vocabulary originally designed for the baseline solution.

	Training and validation				Evaluation on the test set				
	μ_{train}	σ_{train}	$\mu_{\mathbf{val}}$	σ_{val}	acc	μ_{prec}	μ_{recall}	mcc	macro F1
BigBird (with preprocessing)	**0.9721**	0.0007	**0.9657**	0.0021	**0.9652**	**0.9645**	**0.9652**	**0.9533**	0.8457
BigBird (no preprocessing)	0.9697	0.0003	0.9605	0.0017	0.9631	0.9596	0.9631	0.9504	**0.8574**

Numerous strategies have been proposed for domain adaptation [54] apart from that utilised in this thesis, some of which could be more effective for software logs. Unfortunately, the constraints of this thesis have not permitted exploring all these strategies. The computational demands for training large neural networks, such as neural language models, stretch the experiment feedback cycles for days, thus limiting the possibilities for testing many hypotheses. Therefore, uncovering the full potential of a neural language model for software log classification remains a topic for follow-up study.

6.2 Assessment of the Classification Results

Upon computing the evaluation metrics, unless the scores are perfect, it remains uncertain whether the classifier achieved its potential. This section presents a critical analysis of the classification scores and offers a theoretical explanation for the performance issues observed in the experimental results.

Assessing performance expectations

The data labelling procedure described in Section 4.1 suggests that all log examples have been originally classified based on the occurrence of particular keywords assumed to be unique for each class. If this assumption holds, it implies that the data should be perfectly separable. Based on this premise, the average validation score $\mu_{val} \approx \mathbf{0.967}$ attained by the best classifier instance (see Table 16) cannot be considered excellent.

Conversely, the text data analysis presented in Section 5.1 revealed that removing noisy content exposes a considerable number of duplicate entries in the dataset. Among these duplicates, certain observations have identical text but are assigned to different labels. This finding exposes an intrinsic stochastic component in the dataset, which necessarily leads to a loss in classifier training and validation scores [20].

One possible explanation for this uncertainty could be that the Stage 2 preprocessing procedure destroys the essential information that made these classes separable in the first place. However, detailed analysis together with the client revealed that even the unprocessed log samples were missing the keywords used by the rule-based solution to assign the ground truth labels. In other words, the text samples in the provided dataset were incomplete, which renders the initial separability assumption false. This circumstance was accepted as a plausible situation in real production settings where text information can be insufficient to discriminate between failure categories.

In the presence of a stochastic component in data, it is possible to quantify the empirical risk of misclassification by counting the log text observations x that happen to be assigned to more than one class and that are most likely to be misclassified. Since this discussion seeks to assess the highest expectations for the solution performance, it employs the best theoretically possible estimator to determine the minimum number of unavoidable classification mistakes due to uncertainty in the dataset.

The theory presented in Section 2.3 defined the Bayes estimator (4) as the one attaining the minimum loss on a population of observation pairs $(x, y) \sim p(x, y)$ [21]. Assuming the dataset S is a credible approximation of the log population distribution, it is possible to construct a perfect classifier that minimises uncertainty impact. If all data points in S were composed of unique text data entries, then such a classifier would achieve 100% accuracy, no matter how poorly the data might be clustered. In the presence of uncertainty, such a perfect classifier $h^*(x)$ produces the optimal prediction $\hat{y}$, which minimises the empirical loss $\hat{L}_S(h^*)$ as expressed in the following set of equations:

$$l(\hat{y}, y) = \mathbf{1}_{\hat{y} \neq y} \tag{25}$$

$$BR_S(\hat{y}(x)) = \min_{\hat{y} \in Y} \sum_{y \in Y} l(\hat{y}, y) P_S(y \mid x) \tag{26}$$

$$\hat{L}_S(h^*) = \sum_{x} BR_S(h^*(x)) P_S(x), \tag{27}$$

where x denotes a log instance that occurs more than once in the dataset S and has more than one label $y \in Y$ assigned to it. The probabilities $P_S(x)$ and $P_S(y \mid x)$ are estimated from the dataset S. The BR term defined in (26) is referred to as empirical Bayes risk [85], which quantifies the minimal expected loss for a prediction $h^*(x)$ given the uncertainty. Finally, the loss computed in (27) quantifies an estimate for the empirical Bayes risk on the whole dataset S.

In this study, the empirical Bayes risk of the dataset produced at Stage 2 of the baseline solution has been found to be $\hat{L}_S(h^*) \approx 0.020$, which implies that no classifier using preprocessed text data for the prediction can achieve an accuracy score better than **0.980** on the given dataset. On the one hand, this performance cap appears to be inflicted solely by preprocessing, which strips away important pieces of information, thus making certain observations indistinguishable. On the other hand, if the majority of the removed content is indeed noise, then the uncertainty uncovered with the help of preprocessing and approximated by (27) should also explain the classification errors observed with the unprocessed text data. In order to clarify this issue, this thesis performs a comparative analysis of the mistakes the classifiers tend to make with and without text data preprocessing and attempts to determine

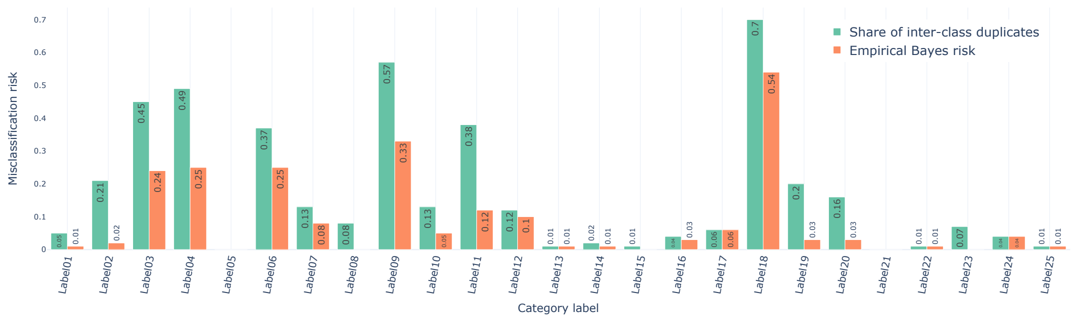


Figure 31: The bar chart presenting the risk of misclassification for each label due to intrinsic uncertainty in the dataset. Green bars represent the share of logs among the label observations that happen to be also assigned to some other label. Red bars quantify the label-specific empirical Bayes risk.

whether the uncertainty can explain these mistakes.

Misclassification due to the uncertainty

For each label, the Bayes risk implied by the uncertainty can be quantified by accumulating the respective error terms from (26). These error terms are the consequence of the best possible classification decisions, given the uncertainty caused by the duplicate logs labelled differently. That is, a classifier is bound to suffer a greater loss in accuracy than that estimated in (27) if its prediction mistakes deviate from those implied by the Bayes risk. Figure 31 presents the Bayes risk component along with the share of duplicate observations computed individually for each label.

As seen from the figure, labels 3, 4, 6, 9, 11, and 18 are at the highest risk of misclassification due to the uncertainty in the dataset. Visual analysis of Figures A1 to A3 suggests that the labels from this risk group indeed tend to be frequently misclassified. Comparing these figures to Figure A4 reveals a considerable similarity of the pairwise classification errors with that implied by the uncertainty. Moreover, the misclassification pattern observed in Figure 31 correlates highly with those obtained from the experimental results presented in Section 5.

The correlation analysis is presented in Table 18. As seen from the first two rows of the table, there is a moderate to the high correlation of the classification mistakes observed in the experiments with those implied by the uncertainty in text data and the Bayes risk. Understandably, the correlation is a bit higher for the results obtained with the preprocessed text data (last two columns). Nevertheless, even for those without preprocessing (first three columns), the correlation is considerably stronger than could be explained by a random chance. That is, under the assumption of no positive correlation, the t-test yields a p-value of less than 0.0008 for the correlation score as moderate as 0.621 and even less so for larger scores.

Table 18: Correlation coefficients measuring the linear relationship between the misclassification pattern implied by the empirical risk analysis and the classification error distribution exhibited by the best performing classifiers (with and without text data preprocessing). Last 3 rows present pairwise correlations for the solutions that use no text preprocessing.

	Not preprocessed text data			Preprocessed text data	
	SVM (Text+metadata)	SVM (Text only)	BigBird	SVM (Text+metadata)	SVM (Text only)
Bayes risk	0.640	**0.711**	**0.621**	0.701	**0.782**
Duplicate share	0.673	**0.794**	**0.746**	0.742	**0.827**
SVM (Text+metadata)	1.000	0.818	0.741	0.885	**0.854**
SVM (Text only)	0.818	1.000	0.918	0.762	**0.844**
BigBird	0.741	0.918	1.000	0.664	**0.841**

When analysing the effect of metadata variables, their use seems to produce correlation scores that are only marginally lower compared to those produced by the text-only variants (see the first two and the last two columns of the first two table rows). This observation agrees with the results of the comparative analysis performed earlier in Section 6.1 that revealed a relatively minor impact of the metadata variables on the classifier performance.

Furthermore, when comparing misclassification patterns obtained with and without preprocessing, both tend to accrue similar mistakes, as follows from the pairwise correlation analysis presented in the last three rows of Table 18. The high correlation scores of > 0.84 presented in the last column of the last three rows are the most indicative of this similarity. Such a strong correlation indicates that the uncertainty revealed with the help of preprocessing might indeed be responsible for the same classification mistakes that also occur with the unprocessed text data.

Other reasons for misclassification

Although the intrinsic uncertainty in text data explains many cases of misclassification, it is not the only reason for the observed classification mistakes. According to the theory of supervised learning presented in Section 2.3, a classifier learned by the ERM rule (2) is also subject to approximation and estimation errors defined in (6), which are added on top of the minimum loss estimated by (27). Therefore, these two error terms can explain any classification mistake that cannot be attributed to the Bayes risk [8], [20].

For example, the classifier bias towards labels overrepresented in the dataset is likely to contribute to the *estimation error* term because this bias can be alleviated by tuning the classifier parameters, e.g., by balancing the observation weights. However, when comparing the linear and RBF kernel SVM classifiers, the lower score of the former is likely due to the *approximation error* caused by the insufficient expressive power of the linear hypothesis class with respect to the data used in the experiments.

Generally, it is rather challenging to quantify the individual error terms when analysing ML model performance. For practical purposes, it is more important to ensure that these errors are well balanced in terms of the *bias-complexity trade off* [20]. For this reason, model evaluation and selection, described along with Stage 4 results in Sections 5.1 to 5.2, relied on the bias-complexity trade-off principle when identifying the best-performing classifier.

Improving classifier performance

The success of learning a good classifier is largely determined by the quality of information encapsulated in the input data. Careful engineering of this aspect could considerably enhance the performance of an ML model. For this purpose, the main effort should be directed at eliminating the sources of performance loss identified in the discussion above.

To tackle the uncertainty in the data, it is necessary to ensure that the text fragments collected from the field are diverse and include important class-specific information. In addition, critical numerical information in test reports could be represented in the form of numerical variables rather than individual tokens in unstructured text. Such a structured representation of numerical data would enable more sophisticated feature engineering and permit a classifier to automatically extract clues that involve simple numerical reasoning.

To address the estimation and approximation errors, the underlying theory (Section 2.3) implies two main strategies. First, increasing the number of distinct observations, especially for the underrepresented classes, should help in reducing the estimation error, while securing a lower risk of misclassification in accordance with the fundamental result in (7). Second, achieving better clustering of classes in the vector space representation would reduce the approximation error and permit using a classifier of lower expressive power, which should lower the generalisation upper bound, defined in (8).

Taken together, these approaches would provide a powerful combination that is bound to improve classifier performance.

7 Conclusion

This thesis has developed a solution for classifying failed test reports generated by the test automation framework of the client organisation. This solution adopted a learning-based approach to address common problems affecting the client's rule-based solution. While the rule-based solution relies on a collection of handcrafted rules to discriminate between classes, the learning-based solution utilises the entire corpus of classification examples to automatically learn an effective discriminator function.

To determine the feasibility of the learning-based solution, this thesis has implemented and evaluated two competing machine learning approaches: a conventional natural language processing (NLP) pipeline and a transfer learning (TL) pipeline employing a neural language model. One conceptual difference between the two approaches lies in the methods used for numerical representation of text data. The NLP pipeline is flexible in the choice of text data encoding techniques but relies on manual vocabulary management through text data analysis and preprocessing as a mechanism for improving numerical representation. This approach has been widely used in the research dedicated to software log classification and was hence selected as the baseline solution for this study. The TL pipeline exploits the capacity of pre-trained neural language models to generate contextual vector space representations by utilising its general knowledge about the semantic features of a large vocabulary of tokens learned from multiple corpora of curated text data. This approach has recently become state-of-the-art in NLP and is enticing for its potential to obviate the need for manual vocabulary management.

Each solution implementation consists of multiple software modules corresponding to the respective solution pipeline stages that facilitate data input, analysis and transformation as well as classifier model training and evaluation. The choice of particular classifier models in this thesis relied on the experience of the machine learning community and the practical considerations discussed in Section 4. For the NLP pipeline, the kernelised SVM classifier was selected for its versatility and a proven track record in numerous empirical studies. For the TL pipeline, the BigBird transformer neural network was chosen as a good match for the problem setup and a renowned achiever in the NLP practice.

The experiments showed that both solution candidates can effectively train and evaluate a classifier object on a dataset of labelled examples. The experimental results presented in Section 5 demonstrated high classification accuracy for both solutions approaching closely the theoretical limit estimated in Section 6.2. In addition, the diagnostic tools developed in this thesis permit an extensive input data analysis and provide the means for investigating the classification mistakes. For example, these tools helped to reveal a stochastic component in the provided dataset. Following this finding, Section 6.2 demonstrated the link between the intrinsic uncertainty in the dataset and the loss in classification accuracy.

The evaluation of the candidate solutions has also led to unexpected findings. For instance, the NLP pipeline solution positioned to improve classifier performance through text preprocessing and active vocabulary management achieved its highest performance score using a text encoding method that required no preprocessing at all. Just as curious, the TL pipeline solution expected to alleviate the vocabulary management effort achieved its best results on the preprocessed data using the vocabulary manually designed for the NLP pipeline solution. These peculiar findings could likely be attributed to the specific nature of the software log domain and open new directions for follow-up studies.

Although the results of this thesis do provide credible evidence of the capability of a learning-based approach to succeed in the log classification task, it remains uncertain whether the developed solutions could be a complete substitute for the rule-based solution used by the client. Answering this question requires a series of field experiments with a learning-based solution deployed in the actual software development environment. Generally, the problem of integrating an artificial intelligence solution into the workflow as well as its deployment and evolution present a considerable challenge, which merits a thesis project of its own. This study has done the necessary groundwork to make such a project possible. Section 4.1 of this thesis also presented a concept for integrating a classifier training pipeline into the client's continuous integration infrastructure. This concept could serve as a basis for future work.

References

[1] I. Sommerville, *Software Engineering.* Pearson Education Limited, 2016.

[2] G. Kim, J. Humble, P. Debois, and J. Willis, *The DevOps Handbook.* IT Revolution Press, 2016.

[3] J. Zhu, S. He, J. Liu, *et al.*, "Tools and Benchmarks for Automated Log Parsing," in *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019*, 2019.

[4] S. Vajjala, B. Majumder, A. Gupta, and H. Surana, *Practical Natural Language Processing: A Comprehensive Guide to Building Real-World NLP Systems.* O'Reilly, 2020.

[5] W. Meng, Y. Liu, S. Zhang, *et al.*, "LogClass: Anomalous Log Identification and Classification with Partial Labels," *IEEE Transactions on Network and Service Management*, 2021.

[6] A. Catovic, C. Cartwright, Y. T. Gebreyesus, and S. Ferlin, "Linnaeus: A highly reusable and adaptable ML based log classification pipeline," in *Proceedings - 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI, WAIN 2021*, 2021.

[7] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017.

[8] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning.* Cambridge University Press, 2014.

[9] Anaconda, "2020 State of Data Science," Tech. Rep., 2020.

[10] K. Jing and J. Xu, "A Survey on Neural Network Language Models," *arXiv*, 2019.

[11] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Pearson, 2021.

[12] G. Verweij and A. Rao, "Sizing the prize: What′s the real value of AI for your business and how can you capitalise?" *PwC*, 2017.

[13] E. Hechler, M. Oberhofer, and T. Schaeck, *Deploying AI in the Enterprise.* Apress, 2020.

[14] G. O'Brien, *Digital transformation game plan : 34 tenets for masterfully merging technology and business.* O'Reilly, 2020.

[15] D. Zhang, Saurabh Mishra, E. Brynjolfsson, *et al.*, "Artificial Intelligence Index Report 2021," Tech. Rep., 2021.

[16] H. Zheng, Y.-C. Wang, and P. Molino, "COTA: Improving Uber Customer Care with NLP & Machine Learning," *Uber Engineering*, 2018. [Online]. Available: https://eng.uber.com/cota/.

[17] V. Muthusamy, A. Slominski, and V. Ishakian, “Towards enterprise-ready AI deployments minimizing the risk of consuming AI models in business applications,” in *Proceedings - 2018 1st IEEE International Conference on Artificial Intelligence for Industries, AI4I 2018*, 2019.

[18] L. Surya, “AI and DevOps in information technology and its future in the United States,” *International Journal of Creative Research Thoughts*, 2021.

[19] N. Masri, Y. A. Sultan, A. N. Akkila, *et al.*, “Survey of Rule-Based Systems,” Tech. Rep., 2019.

[20] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning.* The MIT Press, 2018.

[21] O. Simeone, *A brief introduction to machine learning for engineers.* Now Publishers, 2018.

[22] C. C. Aggarwal, *Data Mining: The Textbook.* Springer International Publishing, 2015.

[23] P. Duboue, *The Art of Feature Engineering.* Cambridge University Press, 2020.

[24] C. C. Aggarwal, *Machine learning for text.* Springer International Publishing, 2018.

[25] A. Nielsen Michael, *Neural Networks and Deep Learning.* Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html.

[26] F. Chollet, *Deep learning with Python.* Simon and Schuster, 2021.

[27] D. Jurafsky and J. Martin, *Speech & language processing.* Prentice Hall, 2000.

[28] Y. Goldberg, *Neural network methods for natural language processing.* Morgan & Claypool Publishers, 2018.

[29] A. Clark, C. Fox, and S. Lappin, *The Handbook of Computational Linguistics and Natural Language Processing.* Wiley-Blackwell, 2010.

[30] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” in *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, 2013.

[31] Q. Le and T. Mikolov, “Distributed representations of sentences and documents,” in *31st International Conference on Machine Learning, ICML 2014*, 2014.

[32] M. E. Peters, M. Neumann, M. Iyyer, *et al.*, “Deep contextualized word representations,” in *NAACL HLT 2018 - 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 2018.

[33] D. Cer, Y. Yang, S. y. Kong, *et al.*, “Universal sentence encoder,” *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2018.

[34] W. Meng, Y. Liu, F. Zaiter, *et al.*, "LogParse: Making Log Parsing Adaptive through Word Classification," in *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, 2020.

[35] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, "Self-attentive classification-based anomaly detection in unstructured logs," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2020.

[36] D. Jayathilake, "A Mind Map Based Framework for Automated Software Log File Analysis," *International Conference on Software and Computer Applications IPCSIT*, 2011.

[37] D. Jayathilake, "Towards structured log analysis," in *JCSSE 2012 - 9th International Joint Conference on Computer Science and Software Engineering*, 2012.

[38] W. Meng, Y. Liu, Y. Zhu, *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *IJCAI International Joint Conference on Artificial Intelligence*, 2019.

[39] W. Meng, Y. Liu, Y. Huang, *et al.*, "A Semantic-aware Representation Framework for Online Log Analysis," in *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, 2020.

[40] J. Wang, Y. Tang, S. He, *et al.*, "LogEvent2vec: LogEvent-to-vector based anomaly detection for large-scale logs in internet of things," *Sensors (Switzerland)*, 2020.

[41] S. R. Wibisono and A. I. Kistijantoro, "Log Anomaly Detection Using Adaptive Universal Transformer," in *Proceedings - 2019 International Conference on Advanced Informatics: Concepts, Theory, and Applications, ICAICTA 2019*, 2019.

[42] O. Ulku, N. Gozuacik, S. Tanberk, M. A. Aydin, and A. H. Zaim, "Software Log Classification in Telecommunication Industry," in *2021 6th International Conference on Computer Science and Engineering (UBMK)*, IEEE, 2021.

[43] D. Sharma, V. Wason, and P. Johri, "Optimized Classification of Firewall Log Data using Heterogeneous Ensemble Techniques," in *2021 International Conference on Advance Computing and Innovative Technologies in Engineering, ICACITE 2021*, 2021.

[44] R. Ren, J. Cheng, Y. Yin, *et al.*, "Deep Convolutional Neural Networks for Log Event Classification on Distributed Cluster Systems," in *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*, 2019.

[45] 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 *Proceedings - 2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016*, 2016.

[46] F. Zhuang, Z. Qi, K. Duan, *et al.*, “A Comprehensive Survey on Transfer Learning,” *arXiv*, 2021.

[47] B. Neyshabur, H. Sedghi, and C. Zhang, “What is being transferred in transfer learning?” In *Advances in Neural Information Processing Systems*, 2020.

[48] T. Mikolov, M. Karafiát, L. Burget, C. Jan, and S. Khudanpur, “Recurrent neural network based language model,” in *Proceedings of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH 2010*, 2010.

[49] M. Sundermeyer, R. Schlüter, and H. Ney, “LSTM neural networks for language modeling,” in *13th Annual Conference of the International Speech Communication Association 2012, INTERSPEECH 2012*, 2012.

[50] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[51] T. Wolf, L. Debut, V. Sanh, *et al.*, “Transformers: State-of-the-art natural language processing,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Association for Computational Linguistics, 2020, pp. 38–45.

[52] T. Lin, Y. Wang, X. Liu, and X. Qiu, “A Survey of Transformers,” *arXiv*, 2021.

[53] J. Kaplan, S. McCandlish, T. Henighan, *et al.*, “Scaling Laws for Neural Language Models,” *arXiv*, 2020.

[54] K. S. Kalyan, A. Rajasekharan, and S. Sangeetha, “AMMUS : A Survey of Transformer-based Pretrained Models in Natural Language Processing,” *arXiv*, 2021.

[55] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 2019.

[56] Y. Liu, M. Ott, N. Goyal, *et al.*, “RoBERTa: A Robustly Optimized BERT Pretraining Approach,” *arXiv*, 2019.

[57] A. Radfort, K. Narasimhan, T. Salimans, and I. Sutskever, “(OpenAI Transformer): Improving Language Understanding by Generative Pre-Training,” *OpenAI*, 2018.

[58] Radford Alec, Wu Jeffrey, Child Rewon, Luan David, Amodei Dario, and Sutskever Ilya, “Language Models are Unsupervised Multitask Learners,” *OpenAI Blog*, 2019.

[59] T. B. Brown, B. Mann, N. Ryder, *et al.*, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems*, 2020.

[60] I. Beltagy, K. Lo, and A. Cohan, “SCIBERT: A pretrained language model for scientific text,” in *EMNLP-IJCNLP 2019 - 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Proceedings of the Conference*, 2020.

[61] Van Rossum G and Drake FL., *Python 3 Reference Manual.* 2019.

[62] B. Bengfort, R. Bilbro, and T. Ojeda, *Applied text analysis with Python : Enabling Language-Aware Data Products with Machine Learning.* O Reilly, 2018.

[63] V. Bewick, L. Cheek, and J. Ball, *Statistics review 8: Qualitative data - Tests of association*, 2004.

[64] M. Greenacre, *Correspondence Analysis in Practice - Second edition.* Chapman and Hall/CRC, 2007.

[65] The pandas development team, *Pandas-dev/pandas: Pandas*, 2020.

[66] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, *Array programming with NumPy*, 2020.

[67] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, “SciPy 1.0: fundamental algorithms for scientific computing in Python,” *Nature Methods*, 2020.

[68] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, 2011.

[69] Plotly Technologies Inc., *Collaborative data science*, 2015. [Online]. Available: https://plot.ly.

[70] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent Dirichlet allocation,” *Journal of Machine Learning Research*, 2003.

[71] S. Bird, E. Loper, and E. Klein, *Natural Language ToolKit (NLTK) Book.* O’Reilly Media Inc, 2009.

[72] R. Rehurek and P. Sojka, “Gensim–python framework for vector space modelling,” *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 2011.

[73] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, “Do we need hundreds of classifiers to solve real world classification problems?” *Journal of Machine Learning Research*, 2014.

[74] Y. Ma and G. Guo, *Support vector machines applications.* Springer International Publishing, 2014.

[75] D. Tomar and S. Agarwal, “A comparison on multi-class classification methods based on least squares twin support vector machine,” *Knowledge-Based Systems*, 2015.

[76] S. Boyd and L. Vandenberghe, *Convex Optimization.* Cambridge University Press, 2004.

[77] M. Grandini, E. Bagli, and G. Visani, “Metrics for multi-class classification: An overview,” *arXiv*, 2020.

[78] C. C. Chang and C. J. Lin, "LIBSVM: A Library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, 2011.

[79] Intel, *Intel(R) Extension for Scikit-learn**. [Online]. Available: https://github.com/intel/scikit-learn-intelex.

[80] M. Zaheer, G. Guruganesh, A. Dubey, *et al.*, "Big Bird: Transformers for Longer Sequences," *arXiv*, 2020.

[81] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019.

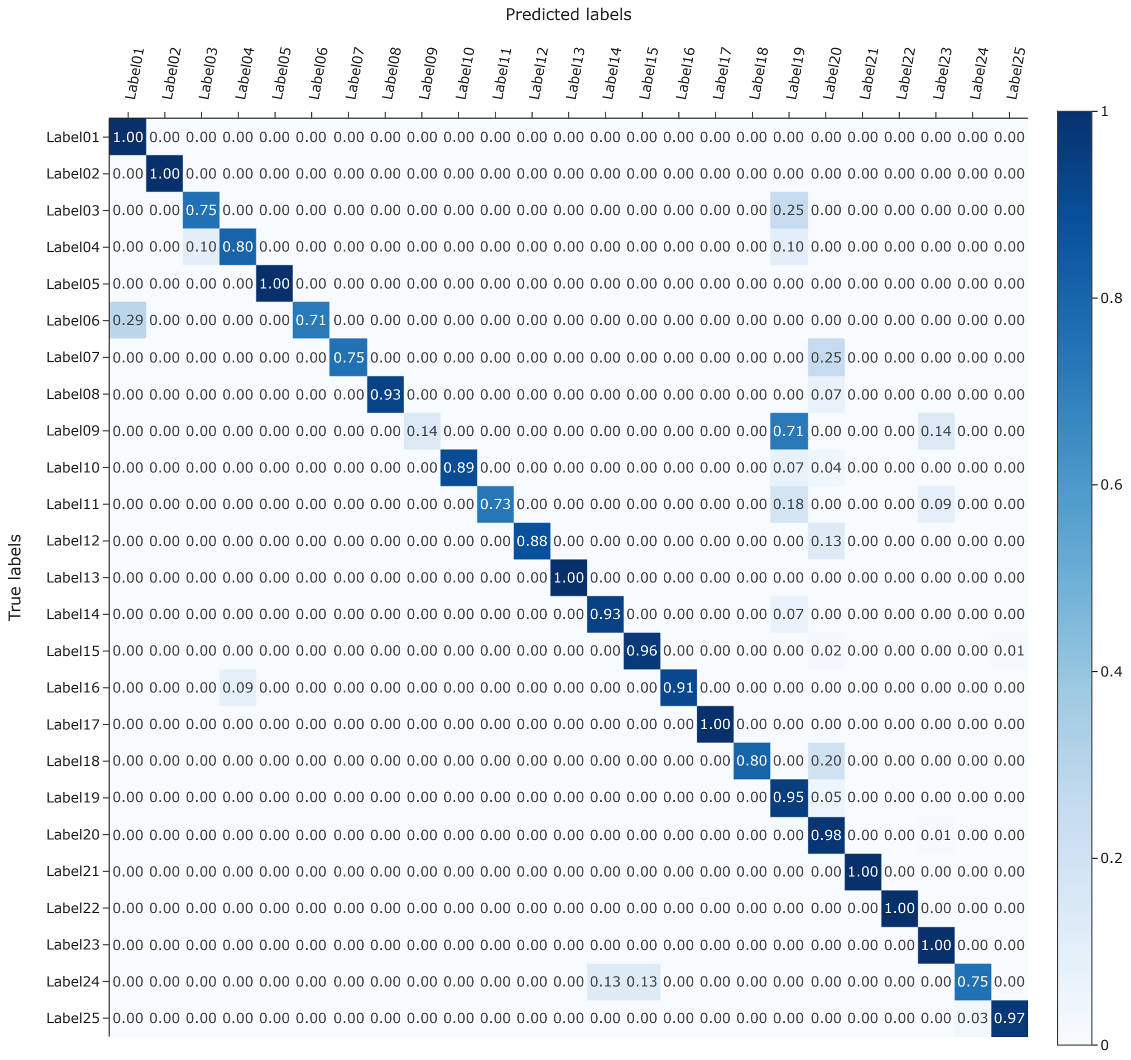
[82] T. C. Rajapakse, *Simple Transformers*, 2019. [Online]. Available: https://github.com/ThilinaRajapakse/simpletransformers.

[83] L. Biewald, "Experiment Tracking with Weights & Biases," *Software available from wandb. com*, 2020.

[84] E. Wallace, Y. Wang, S. Li, S. Singh, and M. Gardner, "Do NLP Models Know Numbers? Probing Numeracy in Embeddings," *arXiv*, 2019.

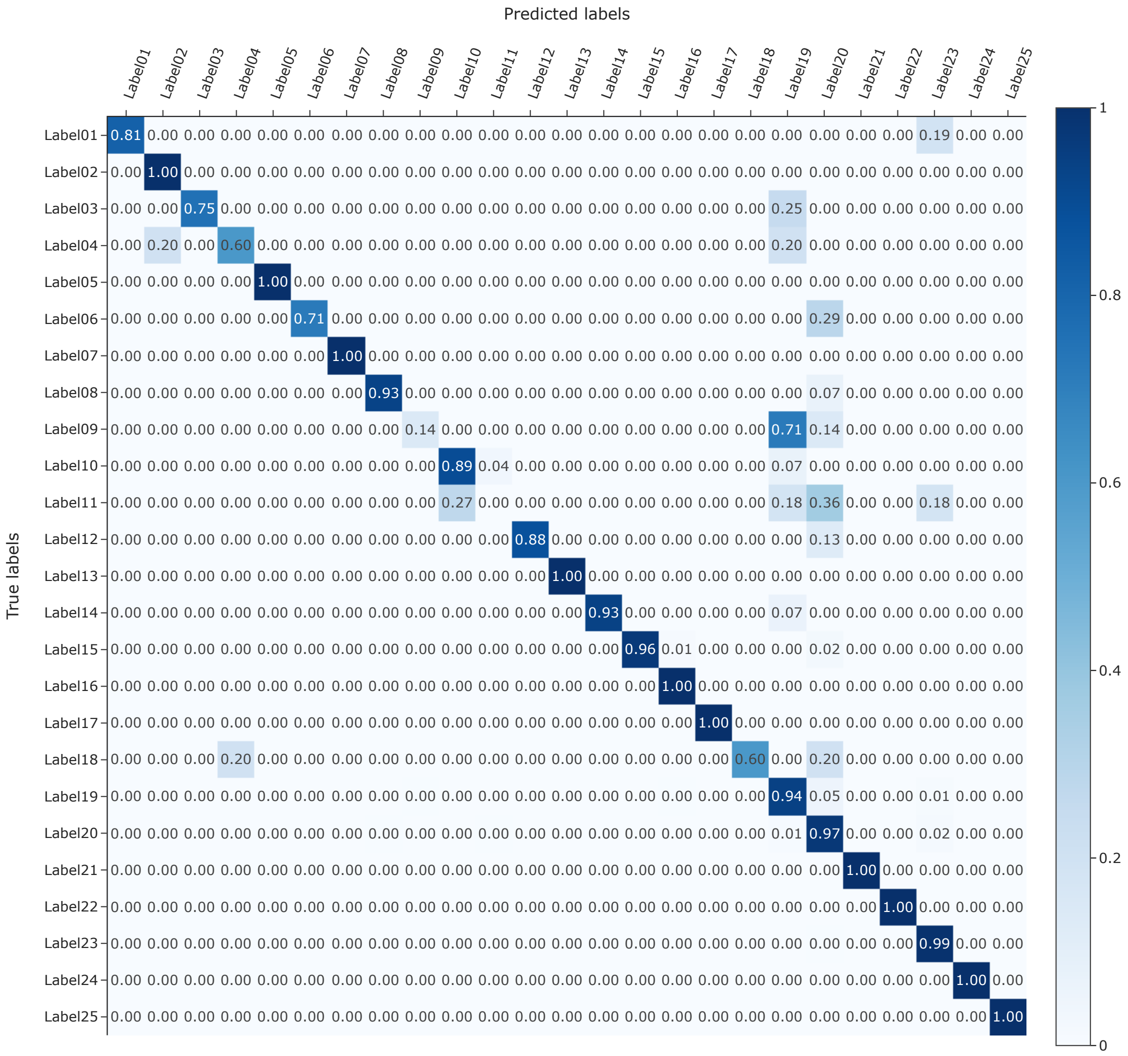
[85] V. Premachandran, D. Tarlow, and D. Batra, "Empirical minimum bayes risk prediction: How to extract an extra few % performance from vision models with just three more parameters," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2014.

A Appendix 1: Confusion Matrices



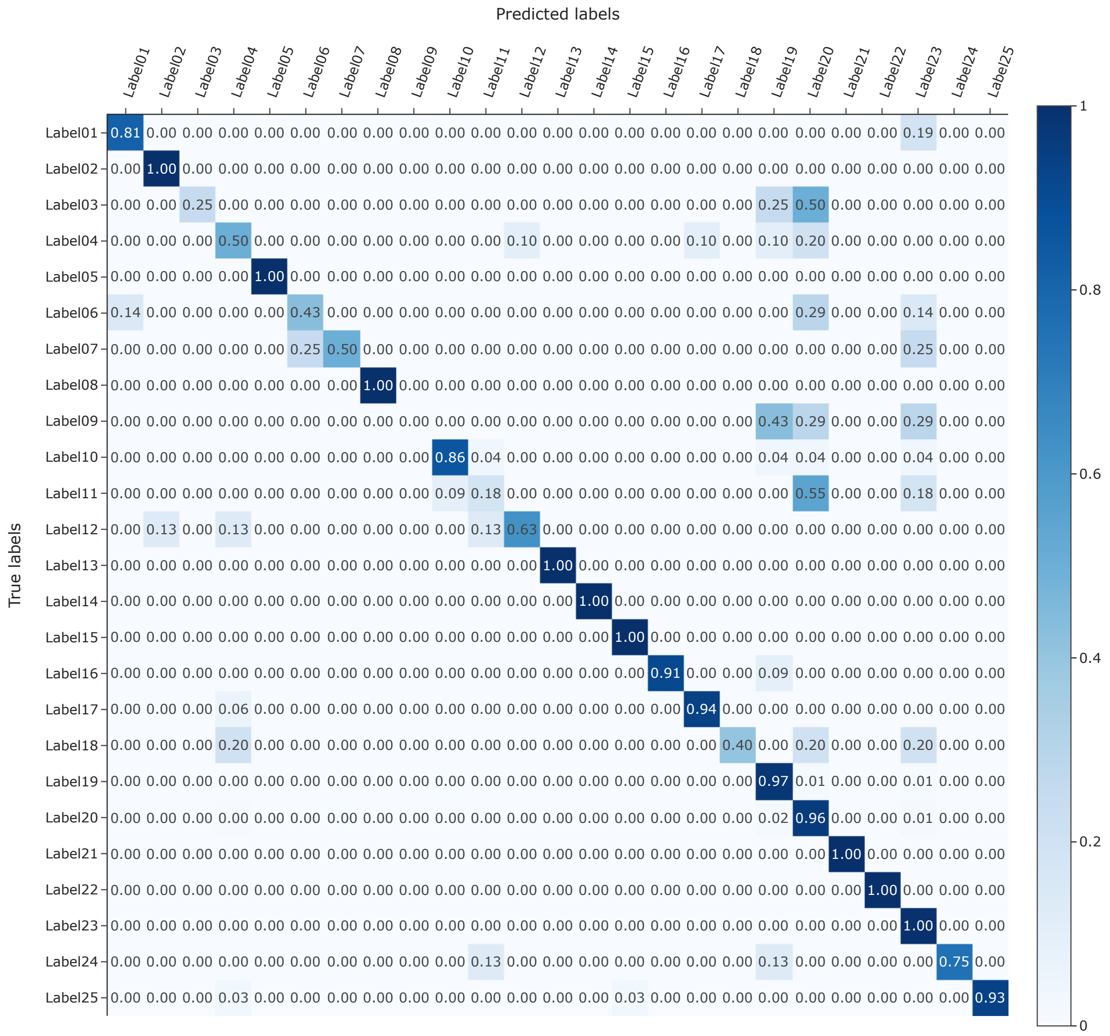
	Label01	Label02	Label03	Label04	Label05	Label06	Label07	Label08	Label09	Label10	Label11	Label12	Label13	Label14	Label15	Label16	Label17	Label18	Label19	Label20	Label21	Label22	Label23	Label24	Label25
Label01	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label02	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label03	0.00	0.00	0.75	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00
Label04	0.00	0.00	0.10	0.80	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00	0.00	0.00
Label05	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label06	0.29	0.00	0.00	0.00	0.00	0.71	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label07	0.00	0.00	0.00	0.00	0.00	0.00	0.75	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.00	0.00
Label08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.93	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.00
Label09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.71	0.00	0.00	0.00	0.14	0.00	0.00
Label10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.89	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.04	0.00	0.00	0.00	0.00	0.00
Label11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.73	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.18	0.00	0.00	0.00	0.09	0.00	0.00
Label12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.88	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.00	0.00	0.00	0.00	0.00
Label13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.93	0.00	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.00
Label15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.96	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.01
Label16	0.00	0.00	0.00	0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.91	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label18	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.80	0.00	0.20	0.00	0.00	0.00	0.00	0.00
Label19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.95	0.05	0.00	0.00	0.00	0.00	0.00
Label20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.98	0.00	0.00	0.01	0.00	0.00
Label21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
Label22	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
Label23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Label24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.75	0.00
Label25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.97

Figure A1: Confusion matrix summarising the best performing SVM classifier predictions on the unprocessed *Word2Vec* test set (text+metadata). The values represent the share of observations of the 'True labels'.



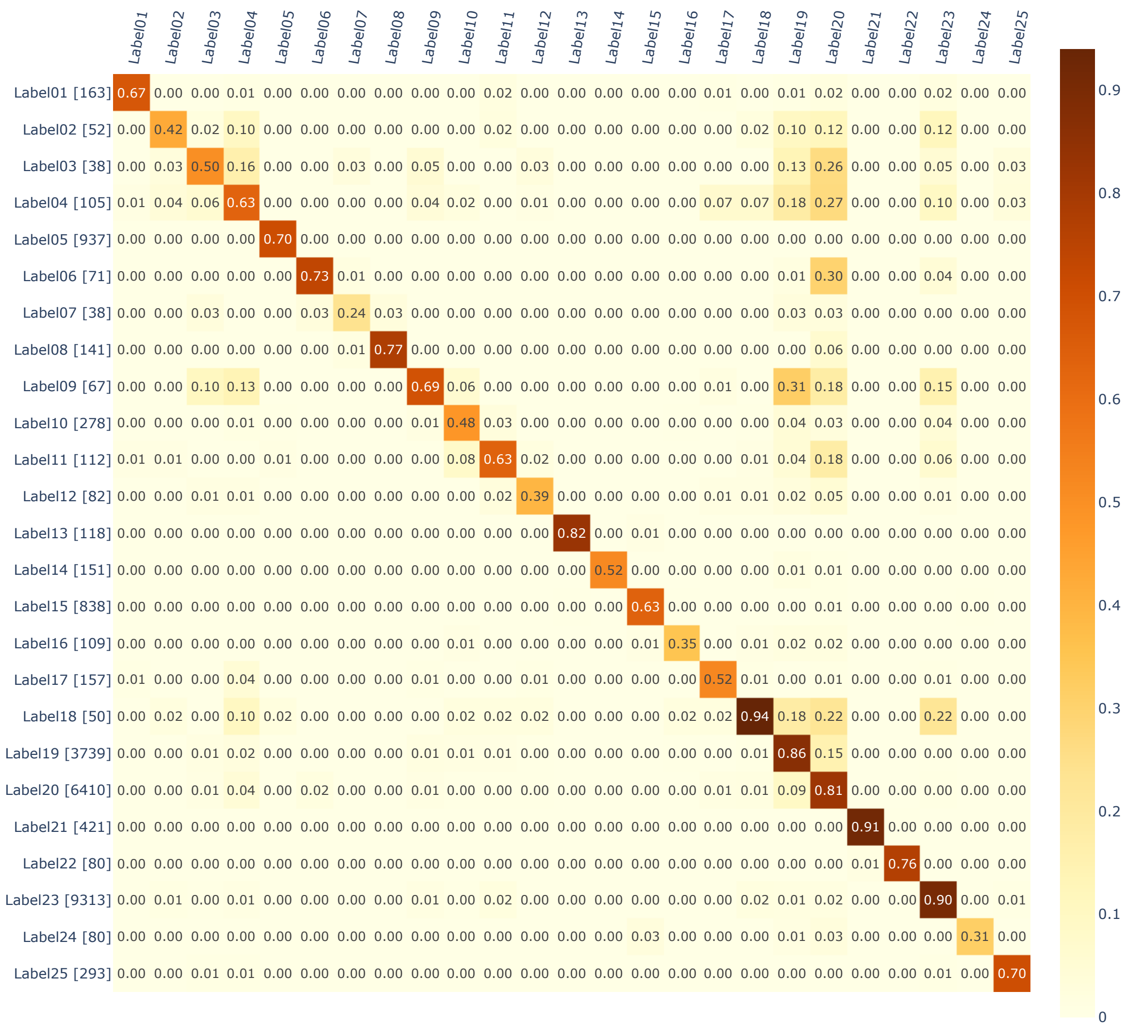
	Label01	Label02	Label03	Label04	Label05	Label06	Label07	Label08	Label09	Label10	Label11	Label12	Label13	Label14	Label15	Label16	Label17	Label18	Label19	Label20	Label21	Label22	Label23	Label24	Label25
Label01	0.81	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.19	0.00	0.00
Label02	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label03	0.00	0.00	0.75	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00
Label04	0.00	0.20	0.00	0.60	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00
Label05	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label06	0.00	0.00	0.00	0.00	0.00	0.71	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.29	0.00	0.00	0.00	0.00	0.00
Label07	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.93	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.00
Label09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.71	0.14	0.00	0.00	0.00	0.00	0.00
Label10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.89	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.00
Label11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.27	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.18	0.36	0.00	0.00	0.18	0.00	0.00
Label12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.88	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.00	0.00	0.00	0.00	0.00
Label13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.93	0.00	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.00
Label15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.96	0.01	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00
Label16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label18	0.00	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.60	0.00	0.20	0.00	0.00	0.00	0.00	0.00
Label19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.94	0.05	0.00	0.00	0.01	0.00	0.00
Label20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.97	0.00	0.00	0.02	0.00	0.00
Label21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
Label22	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
Label23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.99	0.00	0.00
Label24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
Label25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Figure A2: Confusion matrix summarising the best performing BigBird classifier on the unprocessed test set (Text only). The values represent the share of observations of the 'True labels'.



	Label01	Label02	Label03	Label04	Label05	Label06	Label07	Label08	Label09	Label10	Label11	Label12	Label13	Label14	Label15	Label16	Label17	Label18	Label19	Label20	Label21	Label22	Label23	Label24	Label25
Label01	0.81	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.19	0.00	0.00
Label02	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label03	0.00	0.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.50	0.00	0.00	0.00	0.00	0.00
Label04	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00	0.10	0.00	0.10	0.20	0.00	0.00	0.00	0.00	0.00
Label05	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label06	0.14	0.00	0.00	0.00	0.00	0.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.29	0.00	0.00	0.14	0.00	0.00
Label07	0.00	0.00	0.00	0.00	0.00	0.25	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00
Label08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.43	0.29	0.00	0.00	0.29	0.00	0.00
Label10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.86	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.04	0.00	0.00	0.04	0.00	0.00
Label11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.09	0.18	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.55	0.00	0.00	0.18	0.00	0.00
Label12	0.00	0.13	0.00	0.13	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.63	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.91	0.00	0.00	0.09	0.00	0.00	0.00	0.00	0.00	0.00
Label17	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.94	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label18	0.00	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.40	0.00	0.20	0.00	0.00	0.20	0.00	0.00
Label19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.97	0.01	0.00	0.00	0.01	0.00	0.00
Label20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.96	0.00	0.00	0.01	0.00	0.00
Label21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
Label22	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
Label23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Label24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.00	0.00	0.00	0.00	0.75	0.00
Label25	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.93

Figure A3: Confusion matrix summarising the SVM classifier (Text only) predictions on the preprocessed LSI_BoW test set (Text only). The values represent the share of observations of the 'True labels'.



	Label01	Label02	Label03	Label04	Label05	Label06	Label07	Label08	Label09	Label10	Label11	Label12	Label13	Label14	Label15	Label16	Label17	Label18	Label19	Label20	Label21	Label22	Label23	Label24	Label25
Label01 [163]	0.67	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.02	0.00	0.00	0.02	0.00	0.00
Label02 [52]	0.00	0.42	0.02	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.10	0.12	0.00	0.00	0.12	0.00	0.00
Label03 [38]	0.00	0.03	0.50	0.16	0.00	0.00	0.03	0.00	0.05	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.26	0.00	0.00	0.05	0.00	0.03
Label04 [105]	0.01	0.04	0.06	0.63	0.00	0.00	0.00	0.00	0.04	0.02	0.00	0.01	0.00	0.00	0.00	0.00	0.07	0.07	0.18	0.27	0.00	0.00	0.10	0.00	0.03
Label05 [937]	0.00	0.00	0.00	0.00	0.70	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label06 [71]	0.00	0.00	0.00	0.00	0.00	0.73	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.30	0.00	0.00	0.04	0.00	0.00
Label07 [38]	0.00	0.00	0.03	0.00	0.00	0.03	0.24	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.03	0.00	0.00	0.00	0.00	0.00
Label08 [141]	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.77	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.00
Label09 [67]	0.00	0.00	0.10	0.13	0.00	0.00	0.00	0.00	0.69	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.31	0.18	0.00	0.00	0.15	0.00	0.00
Label10 [278]	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.01	0.48	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.03	0.00	0.00	0.04	0.00	0.00
Label11 [112]	0.01	0.01	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.08	0.63	0.02	0.00	0.00	0.00	0.00	0.00	0.01	0.04	0.18	0.00	0.00	0.06	0.00	0.00
Label12 [82]	0.00	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.39	0.00	0.00	0.00	0.00	0.01	0.01	0.02	0.05	0.00	0.00	0.01	0.00	0.00
Label13 [118]	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.82	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Label14 [151]	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.52	0.00	0.00	0.00	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00
Label15 [838]	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.63	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00
Label16 [109]	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.01	0.35	0.00	0.01	0.02	0.02	0.00	0.00	0.00	0.00	0.00
Label17 [157]	0.01	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.52	0.01	0.00	0.01	0.00	0.00	0.01	0.00	0.00
Label18 [50]	0.00	0.02	0.00	0.10	0.02	0.00	0.00	0.00	0.00	0.02	0.02	0.02	0.00	0.00	0.00	0.02	0.02	0.94	0.18	0.22	0.00	0.00	0.22	0.00	0.00
Label19 [3739]	0.00	0.00	0.01	0.02	0.00	0.00	0.00	0.00	0.01	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.86	0.15	0.00	0.00	0.00	0.00	0.00
Label20 [6410]	0.00	0.00	0.01	0.04	0.00	0.02	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.09	0.81	0.00	0.00	0.00	0.00	0.00
Label21 [421]	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.91	0.00	0.00	0.00	0.00
Label22 [80]	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.76	0.00	0.00	0.00
Label23 [9313]	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.00	0.01	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.01	0.02	0.00	0.00	0.90	0.00	0.01
Label24 [80]	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.01	0.03	0.00	0.00	0.00	0.31	0.00
Label25 [293]	0.00	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.70

Figure A4: Matrix presenting the share of duplicate text logs between every pair of labels. Each matrix value represents the ratio of logs in the row-index label that were also found in the column-index label. The values in square brackets of the row indices show the number of data points per label.