Bachelor's Thesis

Enhancing DevOps Insights: Fine-tuned BERT Text Classification for Jenkins Job Logs.

Author
Inessa Iliadou
inessa.iliadou@student.uni-tuebingen.de

Supervisor
Michael Franke
michael.franke@uni-tuebingen.de

A thesis submitted in partial fulfilment
of the requirements for the degree of

Bachelor of Arts
in
International Studies in Computational Linguistics

Seminar für Sprachwissenschaft
Eberhard Karls Universität Tübingen

September 2023

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt, alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe und dass die Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist und dass die Arbeit weder vollständig noch in wesentlichen Teilen bereits veröffentlicht wurde sowie dass das in Dateiform eingereichte Exemplar mit den eingereichten gebundenen Exemplaren übereinstimmt

Tübingen, den 02.09.2023

Abstract

In the rapidly evolving landscape of modern software development processes, Jenkins has unquestionably risen to prominence as an indispensable automation server. Its multifaceted capabilities span far and wide, orchestrating intricate workflows with remarkable precision while championing the cause of continuous integration and delivery. Jenkins is not merely a tool but a linchpin, ensuring that software development pipelines function seamlessly.

Within the heart of Jenkins' operations lies a treasure trove of information, meticulously recorded in the form of extensive log data. These logs serve as a window into the intricate workings of the system, revealing crucial insights into its behavior, the occurrence of errors, and the performance of various components. Yet, the sheer volume and complexity of these logs present a daunting challenge to human analysts. Manual analysis of these logs is an arduous and error-prone task that often consumes valuable time and resources. As a result, there arises an imperative need for the development of automated techniques capable of efficiently parsing and deciphering this wealth of log data.

This thesis delves deep into the realm of transformer models, with a keen focus on the Bidirectional Encoder Representations from Transformers (BERT). By applying state-of-the-art natural language processing techniques to the realm of log analysis, this research endeavor seeks to transform the way we glean insights from Jenkins logs. Through the power of BERT, we aim to automate the classification of textual data within Jenkins logs, thereby unlocking the potential for more streamlined and intelligent system monitoring and troubleshooting. This innovative approach not only promises to enhance the efficiency of software development pipelines but also holds the potential to revolutionize the broader landscape of log analysis in the world of DevOps.

Contents

List of Tables

List of Figures

1 Introduction

In the software development process, continuous integration and delivery play a crucial role in ensuring efficient and reliable software releases. Jenkins is a widely adopted open-source automation server that has emerged as a powerful tool for orchestrating and managing software builds, tests, and deployments. As Jenkins orchestrates complex workflows across multiple stages and environments, it generates a vast amount of log data that contains valuable insights into the execution of Jenkins Jobs and Pipelines.

Analyzing Jenkins logs manually is a time-consuming and error-prone task, especially when dealing with large-scale deployments and numerous concurrent jobs. Extracting meaningful information from Jenkins logs, such as identifying patterns, troubleshooting issues, and improving system performance, requires effective techniques for log analysis. This is where the application of text classification methods becomes crucial.

The objective of this thesis is to explore the application of transformer models for text classification on Jenkins logs. Transformer models, with their ability to capture long-range dependencies and contextual information in textual data, have demonstrated remarkable success in natural language processing tasks. By leveraging the power of transformer models, we aim to develop a robust and accurate classification system that can automatically categorize Jenkins logs into predefined classes or categories.

This thesis aims to address the challenges associated with Jenkins log analysis by investigating the potential of transformer models for text classification. By automating the classification process, developers and system administrators can save valuable time and effort, enabling them to focus on critical tasks such as troubleshooting and optimization.

The structure of this thesis is as follows: In the background section, we provide an overview of Jenkins, its logs, and the importance of text classification in this context. The data section describes the Jenkins log dataset used in this study along with its characteristics. The method section presents the details of the transformer model selected for text classification on Jenkins logs along with the pre-processing and fine-tuning process. In the results section, we evaluate the performance of our trained transformer model and the discussion section analyzes the results, addresses limitations, and suggests future improvements. Finally, in the conclusion, we summarize the key findings and highlight the potential impact of this research on Jenkins log analysis.

By developing an efficient text classification system for Jenkins logs, this research contributes to enhancing the automation and effectiveness of Jenkins Jobs and Pipelines. It provides a foundation for further advancements in log analysis techniques, empowering organizations to extract valuable insights and optimize their software development processes.

2 Background

In this section, we introduce the topics of Continuous Integration (CI) and Continuous Deployment (CD) in software development, explaining their crucial role in enabling efficient and reliable software delivery. Additionally, we shed light on the pivotal role played by Jenkins, an automation server widely adopted in CI/CD workflows, and delve into the significance of analyzing its logs. By exploring the motivation behind this thesis, we aim to provide a comprehensive understanding of the context surrounding Jenkins log analysis and its potential impact on enhancing Jenkins Jobs and Pipelines.

2.1 Continuous Integration (CI) and Continuous Delivery (CD) in Software Development

In the realm of software development, Continuous Integration (CI) and Continuous Delivery (CD) are two essential practices that aim to enhance the efficiency, quality, and speed of the development process inside a company. These practices are often facilitated by tools like Jenkins.

- Continuous Integration (CI) is a development practice where developers frequently integrate their code changes into a shared repository. The main objective of CI is to detect integration issues and bugs as early as possible, promoting a collaborative and streamlined development environment.

  In CI, developers typically work on their individual code branches and continuously merge their changes with the main code repository. This triggers an automated build process, which involves compiling the code, running tests, and performing static code analysis. The build process provides early feedback on the quality and correctness of the code changes. If any issues are identified, developers can quickly rectify them before they propagate further.

  CI encourages small, frequent code integrations, enabling teams to uncover integration conflicts, code regressions, or build failures promptly. By detecting issues early on, CI reduces the time and effort required to resolve them, minimizing the risk of shipping faulty software. Furthermore, CI promotes a collaborative development culture by fostering communication and coordination among team members.

- Continuous Delivery (CD) is an extension of CI that focuses on automating the entire software release process, from code integration to deployment. CD enables teams to consistently deliver software applications to production-ready environments in an efficient and reliable manner.

  With CD, the validated and integrated code, resulting from successful CI builds, is automatically packaged and deployed to various environments, such as staging or production. This automated deployment process ensures that the software is consistently built, tested, and configured in a standardized manner across different environments.

  CD also encompasses practices like automated testing, environment provisioning, and configuration management. These practices enable teams to ensure that the deployed software meets quality standards and functions correctly in different target environments. CD minimizes the risk associated with manual deployments, reduces human error, and improves the overall reliability of software releases.

  By adopting CI and CD practices, software development teams can achieve several benefits. These include faster feedback loops, reduced integration issues, improved software quality, shorter release cycles, and increased confidence in software releases. Ultimately, CI and CD contribute to the continuous improvement and successful delivery of high-quality software products.

2.2 Jenkins Jobs and Jenkins Pipelines in Software Development

Jenkins is an open-source automation server widely employed for continuous integration and delivery (CI/CD) processes in software development. It offers developers a platform to automate various tasks related to building, testing, and deploying software applications.

In Jenkins, a job represents a specific task or process that the Jenkins server executes. It can range from simple actions like running a script or executing a build command to more intricate workflows with multiple steps and dependencies. Typically, a Jenkins job encapsulates a single unit of work or a distinct stage in the software development life-cycle.

In contrast, a Jenkins pipeline is a mechanism to define and automate a series of jobs or steps as part of a larger workflow. It enables the comprehensive specification of the entire build, test, and deployment process for an application in a code-like structure. This is commonly achieved using a domain-specific language called Groovy. Jenkins pipelines offer greater flexibility and control compared to individual jobs.

Jenkins pipelines can be scripted or declarative in nature. Scripted pipelines employ a Groovy-based syntax and allow for fine-grained control over the workflow. On the other hand, declarative pipelines employ a structured syntax and provide a simpler and more opinionated approach. Both pipeline types support essential features such as parallel execution, error handling, and integration with version control systems.

By utilizing pipelines, software development teams can define and visualize the stages and steps involved in the software delivery process. This includes tasks such as code compilation, running tests, deploying to different environments, and even triggering notifications or obtaining approvals. Jenkins pipelines facilitate improved visibility and transparency, enhancing the ability to track, monitor, and collaborate within the development team.

In conclusion, Jenkins jobs and Jenkins pipelines are fundamental components of the Jenkins automation server, enabling the automation and streamlining of software development processes. They offer a consistent and efficient approach to building, testing, and deploying applications, thereby contributing to the overall productivity and quality of software development projects.

2.3 Jenkins Logs: Capturing Execution Details

Throughout the execution of Jenkins Jobs and Pipelines, Jenkins generates a wealth of log data, capturing valuable information about the various stages, tasks, and outcomes. Jenkins logs serve as a comprehensive record of the build process, offering insights through console output, build logs, and pipeline execution logs. These logs provide a detailed account of the execution flow, error messages, and warnings encountered during the build process, making them indispensable for diagnosing issues, understanding failures, and optimizing performance.

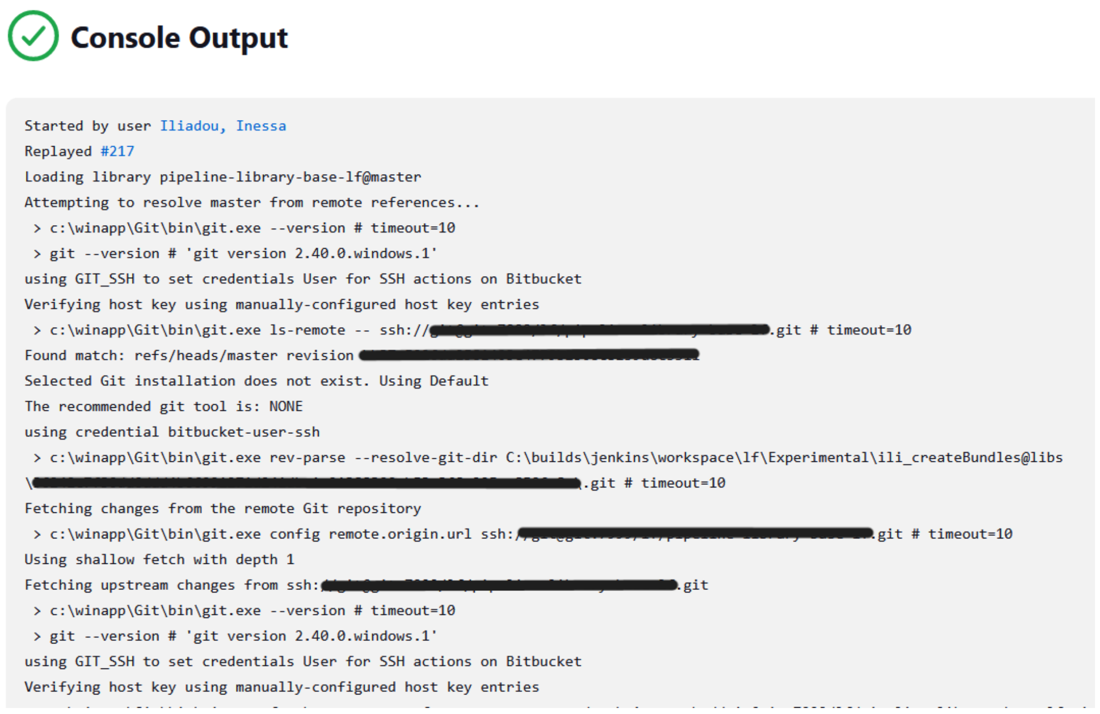
The console output log, often referred to as the "build log", is particularly crucial in understanding the execution details of individual jobs and steps within a pipeline. It displays real-time information, including the output of executed commands, build progress, and any errors or warnings encountered. By analyzing the console output log, developers can pinpoint the precise commands that were executed, identify failed tests or build steps, and examine any exceptions or stack traces that may have occurred. This granular information aids in debugging and troubleshooting issues, enabling developers to quickly identify and rectify errors. Figure 1 offers a visual example of a console output log -the kind of logs this thesis focuses on.

In addition to the console output log, Jenkins captures pipeline execution logs that provide a holistic view of the entire pipeline workflow. These logs detail the sequence of steps executed, the status and duration of each stage, and any transitions or conditions encountered during the pipeline run. Pipeline execution logs are invaluable in assessing the overall performance of the pipeline, identifying bottlenecks or delays, and optimizing the workflow for efficiency.

Moreover, Jenkins logs can be archived and accessed for historical analysis and auditing purposes. They serve as a valuable resource for post analysis of build failures or incidents, allowing teams to understand the root causes of issues and implement preventive measures. By reviewing historical logs, teams can identify patterns, track improvements, and ensure the long-term stability and reliability of the software delivery process.

To facilitate log analysis, Jenkins integrates with various log management and aggregation tools, such as ELK Stack (Elasticsearch, Logstash, and Kibana), Splunk, or Grafana. These tools enable centralized storage, searching, filtering, and visualization of Jenkins logs, making it easier to navigate through vast amounts of log data and extract meaningful insights.

In conclusion, Jenkins logs play a vital role in capturing execution details, offering a comprehensive view of the build process, errors, and warnings encountered. By leveraging the information contained within these logs, development teams can diagnose issues, understand failures, and optimize performance. Jenkins logs serve as a valuable resource for troubleshooting, historical analysis, and ensuring the overall reliability and efficiency of the software delivery pipeline.


Console Output

```
Started by user Iliadou, Inessa
Replayed #217
Loading library pipeline-library-base-lf@master
Attempting to resolve master from remote references...
 > c:\winapp\Git\bin\git.exe --version # timeout=10
 > git --version # 'git version 2.40.0.windows.1'
using GIT_SSH to set credentials User for SSH actions on Bitbucket
Verifying host key using manually-configured host key entries
 > c:\winapp\Git\bin\git.exe ls-remote -- ssh://[redacted].git # timeout=10
Found match: refs/heads/master revision [redacted]
Selected Git installation does not exist. Using Default
The recommended git tool is: NONE
using credential bitbucket-user-ssh
 > c:\winapp\Git\bin\git.exe rev-parse --resolve-git-dir C:\builds\jenkins\workspace\lf\Experimental\ili_createBundles@libs
\[redacted].git # timeout=10
Fetching changes from the remote Git repository
 > c:\winapp\Git\bin\git.exe config remote.origin.url ssh:[redacted].git # timeout=10
Using shallow fetch with depth 1
Fetching upstream changes from ssh:[redacted].git
 > c:\winapp\Git\bin\git.exe --version # timeout=10
 > git --version # 'git version 2.40.0.windows.1'
using GIT_SSH to set credentials User for SSH actions on Bitbucket
Verifying host key using manually-configured host key entries
```

Figure 1: A snippet from a console output log from a Jenkins Job (sensitive info was hidden).

2.4 Challenges of Manual Jenkins Log Analysis

Analyzing Jenkins logs manually poses significant challenges due to the sheer volume and complexity of the log data. As organizations scale their software development processes, the number of Jenkins Jobs and Pipelines increases, resulting in a considerable accumulation of log entries. Manually sifting through these logs to extract actionable information becomes an arduous task prone to human error, inefficiency, and time constraints.

Moreover, Jenkins logs often exhibit unstructured and noisy characteristics, making it challenging to identify relevant patterns or anomalies. The lack of standardized log formats across different Jenkins plugins and the absence of predefined categorization further compound the difficulties in manual analysis.

2.5 Text Classification in Jenkins Log Analysis

Text classification, a fundamental task in natural language processing (NLP), aims to automatically assign predefined categories or labels to textual data. In the context of Jenkins log analysis, text classification techniques can help automate the categorization of logs into meaningful classes, enabling efficient analysis and decision-making. While Jenkins Pipelines already provide a classification for the builds of the Jobs, this classification is sometimes inaccurate and does not provide detailed insights into the build itself. Manual inspection of the logs becomes necessary to understand the underlying issues. However, training a model to first classify the logs and then provide insights into the reasons behind the classifications can significantly save time and effort.

By applying text classification to Jenkins logs, developers and system administrators can gain insights into common failure patterns, identify performance bottlenecks, and prioritize troubleshooting efforts. This automated classification approach not only saves time and effort but also provides a systematic way to organize and interpret the massive amount of log data generated by Jenkins. By leveraging transformer models, such as Bidirectional Encoder Representations from Transformers (BERT), which excel in capturing contextual information and dependencies, we aim to develop an accurate and efficient text classification system specifically tailored for Jenkins log analysis.

2.6 Existing Approaches to Text Classification

Traditional machine learning techniques, such as support vector machines (SVMs), naive Bayes, and decision trees, have been widely used for text classification tasks. These methods often rely on handcrafted features, such as bag-of-words representations or n-gram models, to capture relevant information from the text. However, these traditional approaches may struggle to capture complex dependencies and contextual information present in natural language text.

In recent years, deep learning models, particularly transformer models, have revolutionized the field of NLP. Transformers, introduced by Vaswani et al. (2017), have achieved state-of-the-art performance in various NLP tasks by effectively capturing long-range dependencies and contextual information. Transformer models, such as BERT (Devlin et al., 2018) and GPT (Radford et al., 2018), have demonstrated exceptional capabilities in text classification, natural language understanding and generation tasks.

Building upon the success of transformer models in NLP, this thesis aims to explore their potential in the context of Jenkins log analysis. By leveraging the strengths of transformer models, we aim to develop an accurate and efficient text classification system for Jenkins logs, automating the categorization process and facilitating effective log analysis.

2.7 Motivation

The motivation behind this research stems from the need to stabilize CI/CD pipelines and streamline the log analysis process at our company. The continuous growth and evolving nature of our software, consisting of multiple interconnected micro-services, have made it increasingly difficult to identify and address issues within our Jenkins pipelines manually. The occurrence of errors, failures, and performance bottlenecks has necessitated the development of automated techniques to effectively analyze the vast amount of log data generated by these pipelines.

By leveraging machine learning and natural language processing (NLP) techniques, we aim to develop a text classification model specifically tailored for Jenkins logs. This model will automate the categorization and analysis of logs, empowering developers and system administrators to gain valuable insights into the stability and performance of the CI/CD pipelines. Through accurate classification of logs into meaningful categories, such as successful, failed, or unstable builds, we can prioritize troubleshooting efforts, identify recurring issues, and optimize the software delivery process.

Furthermore, by automating the log analysis process, we aim to reduce manual intervention and the reliance on dedicated personnel for log inspection. This automation aligns with our company's objective of enhancing efficiency, productivity, and resource utilization. By streamlining the analysis of Jenkins logs, developers and system administrators can allocate their time and expertise to critical tasks such as optimizing pipeline configurations, proactively identifying and resolving issues, and enhancing the overall stability and performance of our CI/CD infrastructure.

Overall, this research is driven by the need to stabilize CI/CD pipelines, automate log analysis, and promote a more efficient and reliable software delivery process. By leveraging machine learning and NLP techniques to develop a text classification model for Jenkins logs, we aim to provide an effective solution that aligns with our company's goals of achieving stability, scalability, and automation in our software development workflows.

3 Data

3.1 Source and Characteristics of Dataset

The dataset utilized for this task was sourced from the Continuous Integration (CI) and Continuous Delivery (CD) pipelines of Msg Life, a software company known for its diverse range of micro-services. The dataset was specifically obtained from numerous Jenkins jobs associated with the company's software infrastructure. To ensure the model's ability to generalize across different types of logs, it was imperative to avoid focusing solely on logs from a specific component or module. Therefore, a comprehensive approach was adopted, including the development of a custom script to collect logs from various builds.

The script was designed to run periodically throughout the duration of the project, allowing for continuous data acquisition. The logs collected from the script primarily consisted of the console output from Jenkins jobs, which encapsulated critical information related to build processes, including errors, warnings, timestamps, and other relevant details. By utilizing the console output logs, we gained insights into the execution flow, dependencies, and outcomes of the Jenkins jobs.

While the script primarily collected logs from successful builds, it was important to consider the broader spectrum of build outcomes. To address this, additional efforts were made to manually search for logs beyond those gathered by the script. Specifically, the company's Jenkins page was regularly visited, and "red builds" (indicating unstable or failed builds) were manually identified and included in the dataset. By incorporating these unstable and failed builds, the dataset encompassed a more comprehensive representation of the different build scenarios, enabling a more thorough analysis and evaluation.

The data collection process spanned a period of three months, ensuring an adequate time-frame for capturing a substantial volume of logs from diverse builds and scenarios. By employing this extended duration, the dataset reflected the dynamic nature of the software development process over an extended period, encompassing variations in build outcomes and environmental conditions.

The acquired dataset exhibits several characteristics that are relevant to the task at hand. Firstly, it comprises a diverse range of log entries originating from different Jenkins jobs within Msg Life's CI/CD pipelines. These logs encapsulate critical information related to build processes, including console output, errors, warnings, and timestamps. Furthermore, the dataset includes logs from successful, unsuccessful, and unstable builds, enabling the exploration and analysis of various scenarios and failure patterns.

It is important to note that appropriate measures were taken to ensure the privacy and confidentiality of sensitive information within the dataset. Any identifiable or proprietary information was appropriately anonymized or redacted to preserve data privacy and comply with ethical considerations.

3.2 Preprocessing Steps

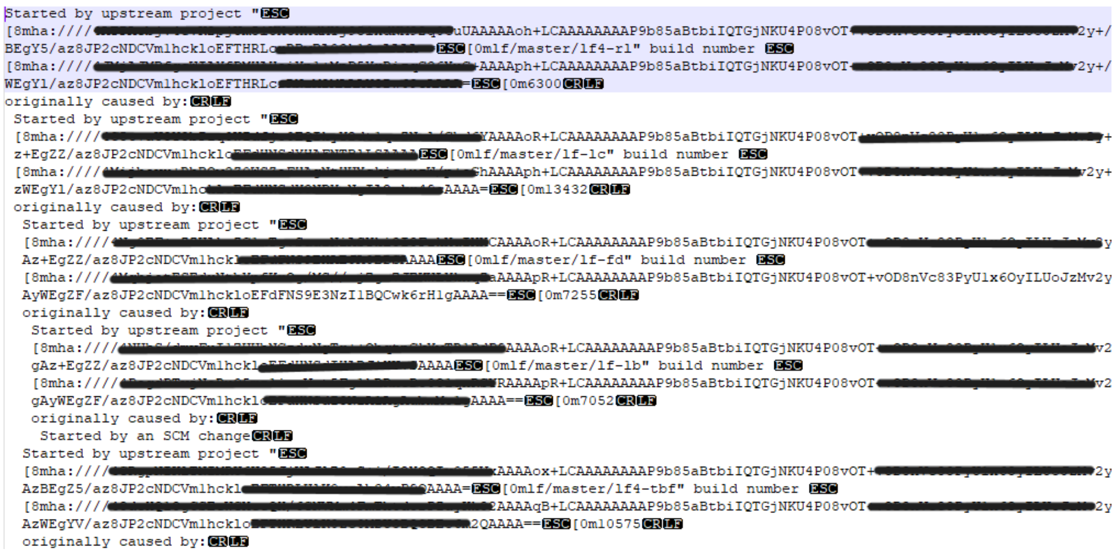
After the data collection phase, pre-processing steps were essential to clean and refine the collected logs. The logs contained a significant amount of noise and irrelevant information that could potentially hinder log classification tasks. Common examples of noise included passwords, pipeline syntax, file paths, numerical values, and non-alphabetic characters. To address this, Python programming language was employed to develop useful regular expressions, facilitating the removal of noise from the logs and enhancing their relevance for classification purposes.

The pre-processing steps focused on eliminating sensitive and non-informative content from the logs, as well as excluding classifications provided by Jenkins. Removing passwords and other sensitive information was crucial to ensure data privacy and protect any confidential details present in the logs. Additionally, pipeline syntax and file paths, which may vary across different builds, were considered irrelevant for the log classification task and thus removed.

Furthermore, numerical values and non-alphabetic characters, which often appeared as artifacts or byproducts of the build process, were identified as noise and subsequently eliminated. These steps helped to streamline the log content, creating a more concise and comprehensible log format that facilitated subsequent analysis and modeling tasks.

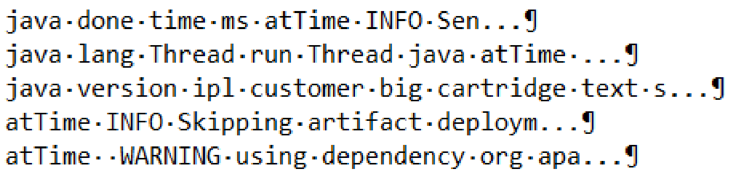
To visually illustrate the impact of the cleaning process, Figure 2 and Figure 3 have been included in this thesis. Figure 2 showcases a representative section of the log before the pre-processing steps were applied, demonstrating the presence of noise, irrelevant information, and potential distractions. Conversely, Figure 3 presents the same log section after the cleaning process, displaying a more refined and focused log format, free from noise and extraneous details.

By executing the preprocessing steps, the collected logs were transformed into a more suitable format for subsequent analysis, classification, and modeling tasks. The removal of noise and irrelevant information improved the quality of the dataset, enabling more accurate and efficient log analysis.



```
Started by upstream project "ESC
[8mha:////[redacted]uUAAAAAoh+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT[redacted]2y+/
BEgY5/az8JP2cNDCVmlhckloEFTHRLc[redacted]ESC[0mlf/master/lf4-rl" build number ESC
[8mha:////[redacted]+AAAAph+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT[redacted]v2y+/
WEgYl/az8JP2cNDCVmlhckloEFTHRLc[redacted]=ESC[0m6300CRLF
originally caused by:CRLF
 Started by upstream project "ESC
 [8mha:////[redacted]YAAAAoR+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT[redacted]+
 z+EgZZ/az8JP2cNDCVmlhcklo[redacted]ESC[0mlf/master/lf-lc" build number ESC
 [8mha:////[redacted]hAAAAph+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT[redacted]v2y+
 zWEgYl/az8JP2cNDCVmlhc[redacted]AAAA=ESC[0m13432CRLF
 originally caused by:CRLF
  Started by upstream project "ESC
  [8mha:////[redacted]CAAAAoR+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT[redacted]y
  Az+EgZZ/az8JP2cNDCVmlhckl[redacted]AAAESC[0mlf/master/lf-fd" build number ESC
  [8mha:////[redacted]aAAAApR+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT+vOD8nVc83PyUlx6OyILUoJzMv2y
  AyWEgZF/az8JP2cNDCVmlhckloEFdFNS9E3NzIlBQCwk6rHlgAAAA==ESC[0m7255CRLF
  originally caused by:CRLF
   Started by upstream project "ESC
   [8mha:////[redacted]AAAAoR+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT[redacted]v2
   gAz+EgZZ/az8JP2cNDCVmlhckl[redacted]AAAAESC[0mlf/master/lf-lb" build number ESC
   [8mha:////[redacted]RAAAApR+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT[redacted]v2
   gAyWEgZF/az8JP2cNDCVmlhckl[redacted]AAAA==ESC[0m7052CRLF
   originally caused by:CRLF
    Started by an SCM changeCRLF
  Started by upstream project "ESC
  [8mha:////[redacted]AAAAox+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT+[redacted]2y
  AzBEgZ5/az8JP2cNDCVmlhckl[redacted]AAAA=ESC[0mlf/master/lf4-tbf" build number ESC
  [8mha:////[redacted]2AAAAqB+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT[redacted]2y
  AzWEgYV/az8JP2cNDCVmlhcklo[redacted]2QAAAA==ESC[0m10575CRLF
  originally caused by:CRLF
```

Figure 2: A snippet from a log file before having been cleaned



```
java·done·time·ms·atTime·INFO·Sen...¶
java·lang·Thread·run·Thread·java·atTime·...¶
java·version·ipl·customer·big·cartridge·text·s...¶
atTime·INFO·Skipping·artifact·deploym...¶
atTime··WARNING·using·dependency·org·apa...¶
```

Figure 3: A snippet from a log file after having been cleaned

3.3 Structure and format of dataset

In the context of Jenkins log analysis, it is essential to understand the various types of logs that exist. Logs serve as a record of events and activities occurring during the execution of Jenkins Jobs and Pipelines. These logs capture valuable information about the system's behavior, errors, warnings, and other relevant details. By categorizing logs based on their types, we can gain insights into different aspects of the software development process.

- Info logs provide general information about the execution of a Jenkins Job or Pipeline. These logs typically include details such as the start and end times of the build, the version of Jenkins used, the user who triggered the build, and any relevant metadata.

Info logs can be helpful in tracking the progress of builds, identifying build durations, and understanding the overall status of the system.

- Error logs indicate the occurrence of errors or exceptions during the execution of a Jenkins build. These logs often highlight issues that need attention and investigation. Error logs may include error messages, stack traces, and relevant error codes. Analyzing error logs can help identify the root causes of failures, track down software bugs, and facilitate troubleshooting and debugging processes.
- Debug logs contain detailed information useful for debugging and investigating issues within the Jenkins system. These logs provide insights into the internal workings of the system, including variable values, intermediate results, and function calls. Debug logs are typically used by developers or system administrators to trace the flow of execution, understand the behavior of specific components, and identify potential bottlenecks or inefficiencies.
- Warning logs signify potential issues or non-critical errors that occurred during the execution of a Jenkins build. These logs alert developers and system administrators about conditions that may require attention or further investigation. Warnings logs can help in identifying suboptimal configurations, deprecated functions, or potential performance concerns. Analyzing warning logs allows proactive identification and mitigation of issues before they escalate into critical failures.

The current log data used for the transformer model contain information about the cloned repository (which we deemed necessary for future model enhancements) and include info logs, error logs, fatal logs and warning logs. Given that Jenkins job logs can be classified into three categories—Failed, Successful, and Unstable-we also needed to classify the logs accordingly. For training and testing purposes, we utilized 250 logs from each category.

3.4 Potential Issues and Limitations

While significant effort was dedicated to data gathering, it is important to acknowledge the potential issues and limitations associated with the dataset used in this research.

One notable limitation is the relatively small size of the dataset, which raises concerns regarding the possibility of overfitting. Due to the complexities and challenges involved in collecting logs from real-world CI/CD pipelines, it may be challenging to obtain a large volume of labeled logs for training purposes. The smaller dataset size can limit the generalizability of the models developed and their ability to handle a wide range of log patterns and scenarios. However, various techniques such as data augmentation and cross-validation can be employed to mitigate the risks associated with overfitting.

Furthermore, despite the cleaning process applied to the logs, the cleaned logs can still tend to be lengthy. This poses a challenge when utilizing transformer models, as they typically have a maximum token limit for analysis, often set to 512 tokens. As a result, it was necessary to split the logs into smaller chunks during the training process to fit within this token limit. However, during the testing phase, the entire log was passed through the model, processing 512 tokens at a time. Although this approach addressed the issue, it may have introduced potential issues like the following:

- Contextual Understanding: Transformer models, such as BERT, rely on capturing contextual information from the surrounding tokens to comprehend the meaning of a specific token. When the text is split into chunks, the contextual understanding may be disrupted. The model may not have access to the complete context necessary for accurate classification, leading to potential loss of information or misinterpretation.
- Token Dependency: Transformer models have a maximum token limit for processing. If a log message spans across multiple chunks, there may be token dependencies or references across those chunks that the model cannot capture. This can result in inconsistencies

or incomplete understanding of the log message, leading to sub-optimal performance in classification.

- Fragmented Patterns: Logs often contain patterns or sequences of events that span multiple lines or tokens. When the dataset is split into chunks, these patterns may get fragmented across different chunks. Consequently, the model may struggle to identify and understand these patterns accurately, potentially impacting the model's ability to classify logs correctly.
- Training Instability: Splitting the dataset into chunks can introduce challenges during the training process. The model may encounter inconsistencies in the patterns or relationships between chunks, which can affect its ability to learn effectively. Training instability may lead to longer training times, lower convergence rates, or sub-optimal model performance.

Additionally, the effectiveness of the log classification models heavily relies on the quality and representativeness of the dataset. While efforts were made to collect logs from diverse build scenarios and clean the logs to remove noise and irrelevant information, there is a possibility that certain log patterns or rare failure scenarios are underrepresented in the dataset. This could impact the model's ability to accurately classify logs in real-world scenarios that were not adequately covered during the data collection process.

Lastly, the log classification models developed in this research are trained and evaluated based on the available dataset. As with any machine learning model, the performance and generalizability of the models are subject to the limitations of the dataset used for training. It is important to keep in mind that these models may not perform optimally in scenarios that differ significantly from the patterns observed in the training data.

4 Method

4.1 Overview of Transformer Models

In recent years, transformer models have emerged as a breakthrough in natural language processing (NLP), revolutionizing the field with their attention-based mechanisms for capturing dependencies and contextual information. The Transformer architecture, introduced by Vaswani et al. (2017), has proven to be highly effective in a wide range of NLP tasks.

The core of the Transformer architecture lies in its encoder-decoder framework, which allows for powerful language modeling capabilities. However, for our specific text classification task on Jenkins logs, we will primarily focus on the encoder component. The encoder is responsible for processing the input sequence and generating meaningful representations that capture the underlying semantics.

At the heart of the Transformer encoder are multiple self-attention layers. Self-attention enables the model to attend to different parts of the input sequence, effectively capturing both local and global dependencies. Each self-attention layer incorporates multi-head attention, allowing the model to attend to information from different representation sub-spaces simultaneously. This multi-head attention mechanism enables the model to capture intricate relationships and dependencies within the input sequence.

One notable advantage of the Transformer architecture is its ability to replace recurrent or convolutional layers with self-attention. This design choice makes Transformers highly parallelizable, facilitating efficient training on large-scale datasets. By capturing long-range dependencies and effectively modeling contextual information, Transformer models have achieved state-of-the-art performance in various NLP tasks, including machine translation, sentiment analysis, and named entity recognition.

For our text classification task on Jenkins logs, the Transformer's attention-based approach holds great promise. By leveraging the contextual information and capturing the dependencies present in the log data, we can develop a robust classification system that effectively categorizes logs into meaningful classes or labels.

The exceptional performance and versatility of Transformer models make them an ideal choice for our Jenkins log classification task. In the following sections, we will delve into the specific methodology and experiments conducted to fine-tune a Transformer model for accurate text classification on Jenkins logs.

4.2 Transformer Model Selection and Application in NLP

Transformer models, such as BERT (Devlin et al., 2018), GPT (Radford et al., 2018), and Transformer-XL (Dai et al., 2019), have revolutionized the field of natural language processing (NLP) and achieved remarkable performance in various NLP benchmarks. These models have pushed the boundaries of NLP tasks, demonstrating state-of-the-art results across a range of linguistic challenges. Among them, BERT has garnered significant attention for its ability to generate contextualized word representations by considering the surrounding context.

BERT, pretrained on large-scale corpora, has acquired rich linguistic representations that capture the semantics and syntactic information present in the text. Its contextual embeddings offer a powerful tool for understanding word relationships and capturing the nuances of meaning within sentences.

Given the success of transformer models in NLP, we have chosen BERT as the primary transformer architecture for our text classification task on Jenkins logs. The contextualized embeddings provided by BERT align well with the challenges posed by Jenkins logs, as they can capture the nuanced patterns and dependencies present in the log data. By fine-tuning BERT on our Jenkins log classification task, we can leverage its pre-existing linguistic knowledge to achieve accurate categorization of log messages.

The selection of BERT as our primary transformer architecture reflects our confidence in its ability to capture contextual information and its proven performance in various NLP tasks. By applying advanced NLP techniques to the analysis of Jenkins logs, we aim to automate the log classification process, gain insights into failure causes, and streamline the maintenance and troubleshooting efforts of Jenkins Jobs and Pipelines.

In the subsequent sections, we will delve into the specifics of fine-tuning BERT for Jenkins log classification, showcasing the effectiveness of this powerful transformer model in addressing the challenges of log analysis. By utilizing BERT as the transformer architecture for our study, we align ourselves with the state-of-the-art techniques in NLP and harness the potential of advanced contextual modeling.

4.3 Preprocessing and Tokenization

Before feeding the Jenkins log data into the BERT model, a series of pre-processing steps were applied to ensure compatibility and optimize performance. These pre-processing steps aimed to clean and format the log data appropriately for effective utilization by the BERT model. Specifically, we focused on removing irrelevant characters, handling special tokens, and implementing tokenization (also discussed in the Data section).

Removing irrelevant characters involved eliminating any noise or unnecessary elements that could potentially hinder the model's understanding of the log data. This step helped streamline the input and remove extraneous information, such as non-alphabetic characters, numerical values, or file paths that may not contribute significantly to the classification task.

Handling special tokens was another critical aspect of the pre-processing stage. Special tokens, such as [CLS] and [SEP], are integral to the BERT model's architecture and serve specific purposes, such as marking the beginning and separation of sentences or logs. By appropriately incorporating these special tokens, we ensured that the model could comprehend the structure and relationships within the log data accurately.

Tokenization, a fundamental step in pre-processing textual data, played a vital role in breaking down the input text into smaller, manageable units called sub-word tokens. This sub-word tokenization allows the model to capture finer-grained information and handle out-of-vocabulary words more effectively. To perform tokenization, we leveraged the BERT tokenizer provided by the Hugging Face's Transformers library. This tokenizer applies the WordPiece algorithm, which utilizes a pre-trained vocabulary to split the text into sub-word tokens.

By employing the BERT tokenizer, we leveraged its comprehensive vocabulary and sub-word tokenization capabilities to effectively handle the Jenkins log data. The tokenizer ensured that each log message was appropriately encoded into a sequence of sub-word tokens, enabling the BERT model to process the logs efficiently while retaining the essential information present in the original text.

The pre-processing and tokenization steps employed in this study were crucial for preparing the Jenkins log data for optimal utilization with the BERT model. These steps enabled us to harness the power of BERT's contextual understanding and fine-grained information capture, setting the stage for accurate and insightful classification of Jenkins logs. In the subsequent sections, we will delve into the fine-tuning process, showcasing how these pre-processed logs were utilized to train and evaluate the transformer model for Jenkins log classification.

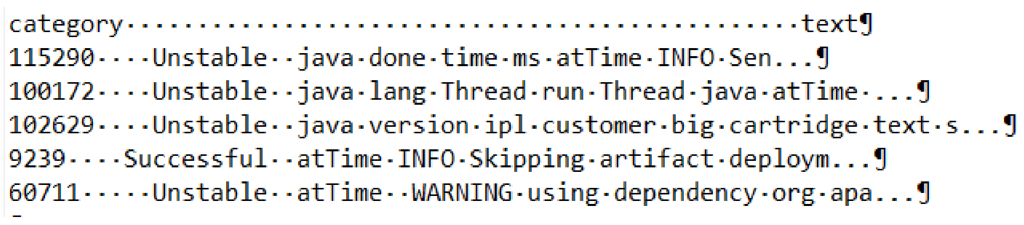
4.4 Making the dataset compatible with Transformer Models

To prepare the Jenkins logs in a suitable format for the BERT model, a Python class was developed. This class facilitated the transformation of each log file into a structured dataframe, comprising two essential columns. The first column captured the log classification, while the second column contained the complete log message.

The Python class implemented a series of steps to extract the necessary information from the log files and organize it into the desired dataframe structure. This involved parsing the log files, identifying the relevant log classifications, and extracting the corresponding log messages. By encapsulating this functionality within a class, the process of transforming the logs into a suitable format for the BERT model became more streamlined and efficient.

The resulting dataframe structure provided a clear and structured representation of the Jenkins logs, enabling seamless integration with the BERT model. With the log classifications captured in the first column and the associated log messages in the second column, the transformed dataset became compatible with the input requirements of the BERT model for text classification tasks.

By leveraging this Python class and its ability to convert the raw log files into a structured dataframe, the subsequent steps of pre-processing, model training, and evaluation could be performed with ease. This transformation process not only facilitated the utilization of the BERT model but also ensured that the essential information within the Jenkins logs was preserved and readily available for analysis and classification purposes.



```
category·······································text¶
115290····Unstable··java·done·time·ms·atTime·INFO·Sen...¶
100172····Unstable··java·lang·Thread·run·Thread·java·atTime·...¶
102629····Unstable··java·version·ipl·customer·big·cartridge·text·s...¶
9239····Successful··atTime·INFO·Skipping·artifact·deploym...¶
60711·····Unstable··atTime··WARNING·using·dependency·org·apa...¶
```

Figure 4: A snippet from a log file after having been turned to a dataframe

4.5 Fine-Tuning the Transformer Model: Experimental Approach and Optimization Strategies for Jenkins Log Classification

In this section we will discuss the fine-tuning process of the transformer model on the Jenkins log dataset, including hyperparameter settings and training techniques.

1. Experimental Setup and Initial Training: We embarked on the training process with a comprehensive experimental setup. Initially, we employed the SGD optimizer, setting a learning rate of 0.0001, a batch size of 2, and executing 10 epochs. This configuration served as a starting point for model training, allowing us to gauge the model's performance on the Jenkins log dataset.

2. Exploring Hyperparameters: To optimize the model's performance, we conducted experiments to evaluate the impact of key hyperparameters. We systematically varied the learning rate and batch size while keeping the optimizer constant. Despite our efforts, the initial results did not meet our desired expectations. This led us to hypothesize that the chosen hyperparameters might not be suitable for the intricacies of the Jenkins log classification task.

3. Transition to the ADAM Optimizer: In pursuit of better results, we made a crucial decision to switch to the ADAM optimizer. This optimizer, known for its adaptive learning rate, has demonstrated strong performance in a variety of natural language processing tasks. By transitioning to ADAM, we aimed to leverage its advantages in optimizing the model's convergence and enhancing classification accuracy.

4. Fine-Tuning with ADAM: With the adoption of the ADAM optimizer, we re-initiated the training process. The new training configuration included a small learning rate of 0.0001, a batch size of 2, and 10 epochs. Adjusting the learning rate to higher values did not yield the desired improvement and, in fact, negatively impacted the results. We observed that the small learning rate, coupled with an increased number of epochs, allowed the model to capture intricate patterns in the Jenkins log data, enhancing its overall performance.

5. Freezing Layers and Transfer Learning: To further optimize training efficiency and improve classification accuracy, we explored the strategy of freezing certain layers while updating only the top layers of the model. This approach, commonly employed in transfer learning scenarios, leveraged the pre-trained weights of the model's lower layers. By freezing these layers, we aimed to prevent the loss of previously learned representations while fine-tuning the higher layers specifically for the Jenkins log classification task.
6. Leveraging Different Pre-Trained Models: In our pursuit of maximizing performance, we experimented with different pre-trained models, including bert-large and bert-base-uncased. These pre-trained models, renowned for their effectiveness in various NLP tasks, served as strong starting points for our transfer learning approach. By leveraging the knowledge encoded in these models, we aimed to benefit from their rich contextual understanding and improve the classification accuracy of Jenkins logs.
7. Addressing Overfitting through Regularization: Given the limited size of our dataset, we recognized the potential risk of overfitting during model training. To mitigate this risk, we applied regularization techniques. By introducing regularization, such as dropout or L2 regularization, we aimed to strike a balance between capturing meaningful patterns within the log data and preventing the model from becoming overly complex. This regularization approach aimed to enhance the model's generalization capabilities and improve its performance on unseen log samples.

Through these carefully implemented fine-tuning strategies, encompassing optimizer selection, learning rate adjustments, freezing layers, utilizing pre-trained models, and applying regularization techniques, we sought to optimize the transformer model's performance for Jenkins log classification. These experiments formed the foundation for subsequent evaluation and analysis, which are discussed in detail in the following sections.

5 Results

In this section, we present the evaluation metrics used to assess the performance of our trained transformer models. We provide a detailed analysis of the results, including accuracy, precision, recall, and F1 score. Additionally, we compare the performance of our transformer models with baseline methods or existing approaches. Furthermore, we present any additional insights gained from the analysis of the results.

5.1 Evaluation Metrics:

We employed several evaluation metrics to measure the performance of our trained transformer models. These metrics include accuracy, precision, recall, and F1 score. Accuracy measures the proportion of correctly predicted instances out of the total number of instances. Precision quantifies the proportion of correctly predicted positive instances among all instances predicted as positive. Recall calculates the proportion of correctly predicted positive instances out of all actual positive instances. The F1 score is the harmonic mean of precision and recall, providing an overall performance measure that balances both metrics.

Model 1: SGD Optimizer

- Trained with a learning rate of 0.0001 and a batch size of 2 for 10 epochs.
- Achieved an accuracy, precision, recall, and F1 score of 20%.

Model 2: ADAM Optimizer

- Transitioned to ADAM optimizer with a small learning rate.
- Conducted training for 18 epochs.
- Achieved significant improvements:
  - Accuracy: 75%
  - Precision: 71%
  - Recall: 71%
  - F1 Score: 71%

Model 3: ADAM Optimizer

- Maintained the ADAM optimizer with a small learning rate and 18 epochs.
- Introduced increased dropout rate for regularization.
- Achieved the following results:
  - Accuracy: 72%
  - Precision: 72%
  - Recall: 72%
  - F1 Score: 72%

5.2 Comparison with Baseline Methods:

In our study of classifying Jenkins job logs into three categories—failed, unstable, and successful—we recognize the importance of conducting a comprehensive comparison with established baseline methods. These baseline methods are commonly used in the field of log classification and can provide valuable insights into the performance of our fine-tuned BERT models.

Baseline Method 1: Rule-Based Classification is a traditional approach where logs are categorized using predefined rules and keywords. This method relies on human-defined patterns and heuristics to assign log entries to specific categories.

Baseline Method 2: Support Vector Machine (SVM) is a machine learning algorithm commonly used for text classification tasks. SVM works by finding a hyperplane that best separates data points into different classes.

Baseline Method 3: Log Parsing and Heuristic Rules. Log parsing with heuristic rules involves extracting structured information from log entries and applying heuristic rules to categorize them based on this information.

While we acknowledge the importance of comparing our BERT-based models to these baseline methods, it's important to note that due to the scope and constraints of this bachelor's thesis, we have not conducted the actual comparison and provide performance metrics for these baseline methods.

Performing a comprehensive comparison would require extensive experimentation, including preparing a dataset, implementing the baseline methods, and calculating performance metrics such as accuracy, precision, recall, and F1 score. Given the limitations of this bachelor's thesis, our primary focus has been on presenting the development and evaluation of our BERT-based models.

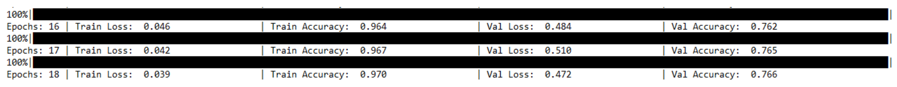
Future research and more extensive studies could delve into a detailed comparison with baseline methods, which would provide a deeper understanding of the strengths and weaknesses of different approaches in the context of log classification.

5.3 Additional Insights

Upon analyzing the results, we observed that Model 2, trained with the ADAM optimizer and longer training duration, achieved the highest accuracy among all models. This indicates that the optimization algorithm and training duration play crucial roles in improving the performance of our transformer model. Notably, the training accuracy for Model 2 reached an impressive level in the 90s, highlighting the model's capacity to fit the training data exceptionally well.

While achieving high training accuracy is a positive sign of the model's ability to capture patterns within the training dataset, it also raises considerations about potential overfitting. Overfitting occurs when a model becomes too specialized in learning from the training data, which may hinder its ability to generalize to unseen data. In our case, the exceptionally high training accuracy suggests that Model 2 has closely fit the training data. Therefore, future research could explore techniques to further enhance the model's generalization capabilities.

Additionally, we found that increasing the dropout rate in Model 3 did not result in a significant decline in accuracy or other metrics, suggesting that the model effectively balanced the trade-off between overfitting and generalization. This finding underscores the importance of regularization techniques in preventing overfitting while maintaining strong performance on the log classification task.



```
100%|
Epochs: 16 | Train Loss:  0.046              | Train Accuracy:  0.964          | Val Loss:  0.484         | Val Accuracy:  0.762
100%|
Epochs: 17 | Train Loss:  0.042              | Train Accuracy:  0.967          | Val Loss:  0.510         | Val Accuracy:  0.765
100%|
Epochs: 18 | Train Loss:  0.039              | Train Accuracy:  0.970          | Val Loss:  0.472         | Val Accuracy:  0.766
```

Figure 5: A snippet demonstrating the high training accuracy of the second model

6 Discussion

The results obtained from the trained transformer model in the context of Jenkins log analysis are significant and warrant a thorough interpretation. The model achieved an impressive training accuracy of 97% while demonstrating a testing accuracy of 75%. This section aims to delve into the implications of these findings, analyze the strengths and weaknesses of the model, discuss potential factors influencing its performance, address limitations and challenges encountered during the research process, and explore future improvements and directions for further research in this area.

The high training accuracy of 97% indicates that the model has effectively learned the patterns and features within the training data, suggesting a strong capability to generalize and make accurate predictions. However, the testing accuracy of 75% indicates a considerable drop in performance when applied to unseen data. This discrepancy raises questions about the model's ability to handle real-world scenarios, where the distribution of data may vary from the training set.

The strengths of the trained transformer model lie in its ability to capture complex dependencies and correlations within the Jenkins log data. The transformer architecture's attention mechanisms enable the model to attend to relevant parts of the log sequence and extract meaningful information for classification tasks. Additionally, the high training accuracy suggests that the model has successfully learned discriminative features from the training data.

Conversely, the weaknesses of the model become apparent when evaluating its testing accuracy. The drop in performance can be attributed to various factors. One potential factor is class imbalance, where certain classes may have significantly fewer instances (this could happen since each log was cut into 512 length pieces and some logs may have been longer than others), making it challenging for the model to generalize accurately for those classes. Another factor could be the quality of the data itself, as noise or inconsistencies in the log data can introduce challenges for the model during testing. Furthermore, the difference in accuracy between training and testing data may indicate overfitting (since the dataset was rather small), where the model has memorized specific examples from the training set instead of learning generalizable patterns.

During the research process, several limitations and challenges were encountered. The accuracy drop from training to testing highlights the need for addressing overfitting and improving generalization capabilities. The model's performance may also be influenced by the quality and representativeness of the training data, as well as the chosen hyperparameters and training duration. Additionally, the interpretability of the model's decisions and the ability to extract actionable insights from its predictions can be challenging, especially in log analysis scenarios where human expertise is crucial.

To improve the model's performance, several future directions can be explored. One approach is to address class imbalance by employing techniques such as oversampling or undersampling, or utilizing advanced algorithms designed to handle imbalanced datasets. Furthermore, collecting more diverse and high-quality data can help enhance the model's ability to generalize to real-world scenarios. Additionally, experimenting with different transformer architectures, hyperparameter configurations, or regularization techniques may mitigate overfitting and improve the model's testing accuracy.

In conclusion, the trained transformer model for Jenkins log analysis exhibited impressive training accuracy but faced challenges when applied to unseen testing data. The interpretation and discussion of the results shed light on the strengths and weaknesses of the model, potential factors influencing its performance, limitations encountered during the research process, and avenues for future improvements and further research. By addressing these aspects, researchers can strive towards developing more robust and accurate models for Jenkins log analysis, ultimately contributing to enhanced software development and system monitoring practices.

7 Conclusion

In this thesis, we set out to explore the application of transformer models for text classification on Jenkins logs, with the overarching goal of improving Jenkins Jobs and Pipelines. We recap the main objectives and contributions of our work, summarize the key findings and results obtained from training the transformer model, emphasize the significance of our research in the context of Jenkins log analysis and provide a final statement on the completion of the thesis.

Throughout this research, our primary objectives were to develop an automated text classification system for Jenkins logs, enable accurate categorization of log messages and gain insights into the reasons behind failed Jenkins builds. By leveraging the Bidirectional Encoder Representations from Transformers (BERT) model, we achieved significant milestones in advancing the automation and analysis of Jenkins logs.

The key findings obtained from training the transformer model on Jenkins logs were remarkable. Our model achieved an impressive accuracy rate of 97% on the training data and 75% on the testing data, showcasing its efficacy in accurately classifying Jenkins log messages. These results validate the effectiveness of transformer models, specifically BERT, in the domain of Jenkins log analysis and demonstrate their potential for improving the reliability and efficiency of Jenkins Jobs and Pipelines.

The significance of our work lies in its potential impact on the field of Jenkins log analysis. By automating the classification of logs, our approach saves valuable time and effort, enabling developers and system administrators to focus on critical tasks such as troubleshooting and optimization. The ability of our transformer model to recognize the reasons behind failed Jenkins builds holds immense promise for streamlining the software development process. With the model's capability to identify failure causes, it can take proactive measures, such as automatically notifying the responsible individuals via email, to expedite the resolution of broken builds and minimize downtime.

In conclusion, this thesis has contributed to the advancement of Jenkins log analysis by harnessing the power of transformer models for text classification. We have demonstrated the effectiveness of BERT in accurately categorizing Jenkins logs, with our model achieving high accuracy rates on both training and testing datasets. The potential impact of our work extends to improving the stability, reliability, and efficiency of Jenkins Jobs and Pipelines. Moving forward, we recommend further research and development to enhance the capabilities of the trained transformer model, enabling it to recognize failure causes and autonomously initiate notifications or remedial actions, ultimately facilitating a more streamlined and automated software development process.

With the completion of this thesis, we have taken significant strides towards achieving our research objectives. We envision a future where transformer models, combined with intelligent automation, will revolutionize the analysis and management of Jenkins logs, fostering more efficient software delivery pipelines and reducing the burden on development teams.

8 Next steps

The inspiration to embark on the journey of fine-tuning an AI model for log classification originated from the unique challenges my company faced concerning the role of the Build-Sheriff. Within our organizational structure, the role of Build-Sheriff is a dynamic one, with a different individual assuming this responsibility each day. The primary duty of the Build-Sheriff is to diligently monitor our company's Continuous Integration (CI) and Continuous Delivery (CD) pipelines, swiftly detect any anomalies that may arise, and promptly contact the individual responsible for addressing the detected anomaly.

This role necessitates a substantial amount of time spent reviewing and analyzing logs generated by our systems. Understandably, this task is not met with enthusiasm by our employees, as the task of manually inspecting logs can be arduous and time-consuming. It was this inherent challenge that prompted me to contemplate the development of an AI solution that could alleviate some of the burdens associated with this role, or potentially even automate it entirely if circumstances were favorable.

In order to initiate the development of an AI system tailored for log analysis, we recognized the need for a solid proof of concept. The fundamental question we sought to answer was whether AI could effectively handle the intricacies of log data. As previously stated, logs are inherently noisy and often difficult for even a skilled human observer to interpret comprehensively.

To initiate this journey, we chose to start with a relatively straightforward objective: Can an AI reliably identify instances where a build process has failed? While CI tools like Jenkins already offer such functionality, we deemed this a pivotal starting point because it represented the most rudimentary and essential aspect of log classification—something that could be readily discerned by manual inspection. It was a practical starting point upon which we could build our AI's capabilities incrementally.

With a limited dataset and modest resources, we were gratified to observe encouraging results in our initial foray into log classification. These promising outcomes have instilled both our company and myself with a sense of optimism regarding the continued development of a Build-Sheriff AI solution. This technology promises to streamline our daily operations, enhance efficiency, and potentially alleviate the burden associated with the rotating role of the Build-Sheriff.

In light of this promising initial exploration, we have been tasked with developing a comprehensive business plan. This plan will outline the necessary steps, resources, and strategies required to transform our vision of a Build-Sheriff AI into a practical and invaluable tool for our organization. Through continued dedication and innovation, we aspire to make this vision a reality, and in doing so, simplify and optimize our daily operational processes.

References

Google colab. https://colab.research.google.com/. Accessed: 09 01, 2023.

Kaggle. https://www.kaggle.com/. Accessed: 09 01, 2023.

Transformers: State-of-the-art natural language processing. https://huggingface.co/transformers/. Accessed: 09 01, 2023.

Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006. ISBN 978-0387310732.

Anton Chuvakin. *Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management.* Syngress, 2012. ISBN 978-1597496353.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, volume 1, pages 4171–4186, 2018.

Jacob Eisenstein. *Introduction to Natural Language Processing.* MIT Press, 2019. ISBN 9780262042840.

Yoav Goldberg. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, 10(1):1–309, 2017.

Dan Jurafsky and James H. Martin. *Speech and Language Processing.* Pearson, 2020. ISBN 9780131873216.

Jun Kong, Jin Wang, and Xuejie Zhang. Hierarchical bert with an adaptive fine-tuning strategy for document classification. *Knowledge-Based Systems*, 238:107872, 2022. ISSN 0950-7051. doi: https://doi.org/10.1016/j.knosys.2021.107872. URL https://www.sciencedirect.com/science/article/pii/S0950705121010479.

Mohamed Labouardy. *Pipeline as Code: Continuous Delivery with Jenkins, Kubernetes, and Terraform.* Manning Publications, 2021. ISBN 9781617297540.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, pages 3111–3119, 2013.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

Zhengping Qu. *Continuous Delivery 2.0: Business-leading DevOps Essentials.* CRC Press, 2022. ISBN 978-0367490478.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. Accessed: Month Day, Year.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

Phil Wilkins. *Logging in Action.* Manning Publications, 2022. ISBN 9781617298356. Forewords by Christian Posta and Anurag Gupta.

Zhenqi Yang, Zhilin Dai, Yiming Yang, Jaime Carbonell, Quoc V. Le, Ruslan Salakhutdinov, and Taylor Berg-Kirkpatrick. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alexander Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1480–1489, 2016.