CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Improving Continuous Integration Feedback Flow

A Design Science Study

Master's thesis in Computer science and engineering

Christian Lind

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Improving Continuous Integration Feedback Flow

A Design Science Study

Christian Lind

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Improving Continuous Integration Feedback Flow
A Design Science Study
Christian Lind

Supervisor: Miroslaw Staron, Department of Computer Science and Engineering
Advisor: Patrik Firek, Zenseact
Examiner: Eric Knauss, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Improving Continuous Integration Feedback Flow
A Design Science Study
Christian Lind
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Continuous integration represents a prevalent practice involving the automated merging of code modifications from various contributors into a unified software project. Despite its widespread adoption, this process often entails considerable time and is susceptible to failures. Consequently, efforts have been directed towards anticipating the outcome of the continuous integration process prior to its initiation. This thesis explores the feasibility of predicting the outcome in near-real-time, leveraging the data accessible within the continuous integration job at that specific moment, employing a design science research approach across three iterative cycles.

Utilizing the design science research approach, the thesis initially delved into the issue by gathering data through interviews and a concise literature review. This process resulted in identifying the problem of delivering improved and swifter feedback to developers. The literature review also unearthed prior efforts aimed at addressing the same issue, prompting an exploration into employing machine learning to forecast build outcomes based on continuous integration (CI) job log data. The outcomes of evaluating various algorithms spurred both empirical and qualitative/quantitative analyses, augmented by interviews with developers at Zenseact.

The primary contribution lies in the crafted artifact itself, a significant addition to the realm of predicting the outcome of continuous integration job builds, serving as a practical solution validated within an industrial setting. This artifact not only introduces innovative resolutions to recognized challenges but also enriches the repository of design science knowledge.

Keywords: continuous integration, machine learning, just-in-time prediction, design science research

Acknowledgements

I would first and foremost like to thank my academic supervisor, Miroslaw Staron, and my industrial supervisor, Patrik Firek, for helping me during the project with any challenges that occurred. I would also like to thank David Friberg for guidance on legal issues and for making the necessary preparations for the thesis. Lastly, I would like to thank the rest of the Overflow team and all the developers at Zenseact who supported me in any way while working on the thesis. Christian Lind,

Gothenburg, June 2024

Contents

List of Figures

List of Tables

1
Introduction

As software repositories become bigger with more and more tests added the time taken for a continuous integration (CI) job to complete continues to increase. After committing changes to a version control system, developers are often eager to promptly receive feedback from the CI job regarding the success or failure of their commit [1]. When a CI job takes over 30 minutes [2] to complete the developer loses focus and the workflow is disrupted. They also desire concise and explanatory data to promptly identify the specific failure that led to the erroneous return of the CI job [3].

To address the aforementioned challenges, predicting the build outcome in CI has emerged as a prominent area of interest. Previous studies have primarily focused on predicting job outcomes based on commit metadata. However, scant attention has been paid to predicting the job outcome while the CI job is actively running. During this phase, additional parameters become available, which were not considered in earlier research efforts. Consequently, leveraging these parameters during job execution should enhance the accuracy of outcome predictions.

The primary objective of this study is to identify and implement enhancements to the developer workflow, particularly focusing on expediting feedback from the CI system. This study will be conducted in collaboration with the observability team at Zenseact, a company dedicated to advancing fully autonomous vehicles. Collaborating with Zenseact offers the advantage of leveraging their established CI infrastructure, along with access to tens of thousands of saved logs from previous CI job runs.

1.1 Problem Description

A developer will often want a fast response as to why the code has failed to be integrated with the CI system. This is because they first want to see why the commit in question failed the tests. CI systems today therefore employ techniques that are supposed to make builds faster using cache, multithreading and optimizing code as much as possible [4]. What all of these methods have in common is that they can break an already working system. Even with all these techniques in a large-scale product, the CI jobs can still take multiple hours to finish [4]. Where eagerness for quick feedback may turn into frustration over a slow completion. The pipeline job may also have some problems which are not that easy to spot for a developer not

that invested into how the CI system works [5]. This can mean a pipeline job that takes longer than usual might cause warning signs for newcomers while in reality, it may for example only be that a new update needs to be installed.

One problem for developers is that a single pipeline job can take multiple hours to complete, with over 40% of all jobs reportedly taking more than 30 minutes [2]. This can lead to long wait times for commits to finally be merged into the repository as each new commit has to go through a new integration process until all stages in the CI job pass. If there are multiple bugs in the code that the developer has not fixed this might require multiple commits in order for all fixes to be applied depending on when the developer finds the bugs and how long it takes to correct them. As a result of long build times developers tend to lose focus and this can hurt productivity and parallel development [6]. Long build times can also lead to more computational resources needed in order to complete the build process before a failure is discovered.

Waiting for a CI build to finish or starting another project on the side while waiting for a CI build to finish can negatively affect the productivity of a developer [7]. To remedy this problem machine learning (ML) algorithms can be used in order to predict the outcome of the build [8]. This can help developers in predicting if a build will fail or pass and thus help the developer know the outcome of the build beforehand. However, this approach in ML cannot help in finding where a failure might occur in the code they have uploaded. This can lead to the developer having to spend a lot of time looking through the code to find the failure. The failure might not be that clear at first before running the pipeline job and getting the result, resulting in significant time wasted by the developer.

Another problem for most developers, a problem is how verbose a log should be. When running all checks in a CI job it is typical to also log every step the CI job takes in order to ensure everything is working accordingly. These logs can for example contain the CMake log from building the program with CMake. While these logs can help the developer find the cause of the build issues, it may not be as clear as one would expect [9]. A more verbose build log may contain unnecessary details making it hard to read [10]. On the other hand, a minimal log may not contain all the necessary information for solving the problem [11].

In large CI systems with a lot of building and testing, it is common to get back a log with megabytes of data [12]. When a pipeline job fails it will indicate which stage the failure occurred in but if that stage has a lot of log data, it can still be very time-consuming for the developer to find the failure [13]. With this, it is also frustrating for the developer to get back a large log showing a failure that is hidden behind thousands of lines of log entries. For failed jobs, it can be challenging to quickly find the reason why the underlying job failed [13]. This results in an increased cognitive load on developers. It can also lead to questions about what caused the pipeline to fail [14]. Developers often look for keywords in the logs which might indicate the failure [14]. But this may be misleading and thus trick the developer into fixing something that is not broken in the first place or completely unrelated

to the real issue.

A CI DevOps engineer would want to find failures in the CI job itself. While the developer only cares about the issue that is specific to his commit. This can create a scenario where the developer must search through large logs to find the right issues for his build. Zampetti et al. [8] mention that their survey reports disagreement on how large a log should be, later remarking that it is almost impossible to find a scenario where a more minimalistic log is better. Each passed or failed CI job is accompanied by a log in textual form. For a pipeline job that compiles C++ libraries, the log is a C++ build log, and a failed job should contain information about why the build failed. For a normal developer working with the C++ code, it might be hard to fully understand how to search the log for the related issue.

Today developers often get to choose where it is relevant to log inside of the code [10]. This can be problematic as developers have different domain knowledge. Which can lead to not properly logging in places where an error can occur. This can in turn lead to logs that contain no information regarding the real problem for why a build failed and thus will be hard to debug [10]. Such a problem can be seen in Listing 1 where Docker has tried to push a new Docker build but failed because of some Python script. Usually, there is an exception shown as to why the Python script has failed but this is not the case in this instance. If a log does not contain the information needed, then it will be impossible for a developer as well as a prediction algorithm to correctly find what caused the build to fail. In this case, it is irrelevant how much log data there is and how fast a developer wants to get the information as the relevant information does not exist.

```
&1|20:52:35.222 Running: docker push hub.docker.com/car_platform/
master/71ce2b95d1d76856eceb84/config:src-2.11
&1|20:52:35.265 !! Error when executing ['/var/lib/agent/pipelines/
src/build/a549-4392-19b57c7915cc/venv/bin/python',
'config/jobs/deliver_car_platform.py'], return_code 1
?1|20:52:35.395 [go] Task status: failed (147597 ms) (exit code: 1)
```

Listing 1: Example of log output with no information on what caused the error.

1.2 Purpose of the study

The purpose of the study is to explore to what extent it is possible to better provide feedback in near-real-time to developers from a CI system. The study will result in a proof of concept that entails simulating a CI system that sends streams of new logs in near-real-time to a model for analysis. This model then provides near-real-time feedback if the build is likely to fail. Giving developers the chance to choose between multiple different ways to implement a feedback flow. Thereafter, the feedback flow is then evaluated based on how developers perceive its usefulness compared to a normal CI workflow. Additionally, qualitative analysis is conducted to assess the accuracy and the MCC score of the algorithm in predicting build outcomes

in near-real-time. These findings are then juxtaposed with developers' expectations regarding prediction accuracy. Furthermore, considerations such as time savings and computational resources are factored in to determine the viability of such a solution. Ultimately, this thesis aims to benefit both developers and researchers by providing insights into optimizing the CI workflow.

1.3 Limitations and Delimitations

The thesis will focus on providing feedback to the developer in near-real-time while the CI job is running. It will therefore not feature any study on providing feedback before or after the job has been run. The feedback will only be provided using already existing ML classifier, meaning no research on new classifiers or such.

In this thesis, multiple combinations of models and classifiers will be evaluated, but it is impossible to evaluate all in this thesis's scope. Classifiers have been chosen based on performance in others work that are working in a similar area in predicting a CI build outcome or classifying logs. The models have been created and chosen based on how log files are typically preprocessed in a similar scenario.

Another limitation is that only CI logs from one company are used. There are multiple jobs used, but all are provided by a company specializing in making autonomous driving for cars. There are more aspects of a car than what Zenseact is building and therefore the way to build those pipelines might look different for other companies. This is also true for any other industry as well when the needs of the company are different.

1.4 Significance of the study

This study contributes both to the academic field as well as practitioners. In academia, it explores a novel area by predicting CI job outcomes in near-real-time using log data generated by a CI system. An area previously only looked at before a job has been started. It identifies various methods for making such predictions and evaluates their effectiveness, ultimately presenting an artifact that underscores the effectiveness inherent in developing such a predictive system.

Contribution towards Zenseact has been made by having a new prediction system set up and ready for use. Numerous CI jobs from Zenseact have undergone testing, and additional ones can be seamlessly integrated. While the contribution to Zenseact surpasses that of other companies due to the accessibility of these tools, the knowledge provided in this report facilitates the replication of similar systems by other organizations. Furthermore, companies can fine-tune various parameters to optimize compatibility with their CI systems.

1.5 Thesis outline

In Chapter 1, a concise overview of the thesis subject was presented, including an exploration of the project's objectives and limitations.

Chapter 2 introduces relevant terminology and concepts for this thesis.

Chapter 3 presents other studies relevant to this thesis and how their findings will be used.

Chapter 4 describes the research methodology used in this study.

Chapter 5 describes the developed artifact and its functionality.

Chapter 6 presents the results and analysis of all iterations.

Chapter 7 argues for the findings and answers the research questions.

Chapter 8 draws conclusions for the thesis.

2
Background

This chapter introduces important concepts that are integral to the workflow. It begins by exploring the fundamentals of machine learning (ML) and the evaluation metrics used throughout the thesis, it then proceeds to first discuss traditional machine learning and later on deep learning (DL) techniques, elucidating their respective functionalities.

2.1 Machine learning

Machine learning (ML) is a branch of artificial intelligence (AI) that empowers computers to acquire intelligence and perform tasks without explicit programming. ML is used to simulate human learning activities [15] for obtaining new information and skills in order to continuously improve knowledge. ML algorithms can be broadly categorized into several types, each tailored to different learning scenarios. Supervised learning involves learning from labeled data, where the algorithm predicts output based on input-output pairs provided during training. In contrast, unsupervised learning explores unlabeled data to uncover hidden structures and patterns. Semi-supervised learning represents a hybrid approach, combining elements of supervised and unsupervised learning, and enabling agents to learn through interactions with environments, respectively.

2.1.1 Data preprocessing

Data encompasses the raw information, whether structured or unstructured, that is fed into a model to enable learning and decision-making. This data can come from various sources, such as databases, sensors, text documents, images, or audio recordings. However, data quality and quantity play crucial roles in determining the model's effectiveness and reliability.

Data preprocessing is often the initial step in preparing the data for training with an ML classifier. This involves tasks like cleaning the data to remove noise and inconsistencies, handling missing values, and transforming the data into a format suitable for the model [16]. Removing noise refers to the process of eliminating irrelevant or unwanted information from the dataset. Noise can manifest in various forms such as outliers, errors, or inconsistencies in the data. Removing noise is crucial because it can adversely affect the performance and accuracy of ML classifiers by introducing unnecessary variability or bias.

Additionally, features may need to be extracted or engineered to enhance the classifier's ability to learn relevant patterns and relationships within the data. Features represent individual measurable properties or characteristics of the data used as input for an ML model. Features are essential components of the dataset that provide information to the classifier, allowing it to learn patterns and make predictions. In the end the goal of data preprocessing is to find the best set of features for the classifier used [17]. The two most relevant techniques for this thesis are addressing data imbalance and handling outliers.

Addressing data imbalance
In classification problems, imbalanced datasets occur when one class of the target variable is significantly more prevalent than others [18]. Data cleaning may involve techniques such as resampling (oversampling minority class or undersampling majority class) to address data imbalance and improve the performance of the model. Oversampling involves increasing the number of instances in the minority class by randomly replicating them or generating synthetic samples. Undersampling involves reducing the number of instances in the majority class by randomly selecting a subset of instances. This can help balance the class distribution, but it may also result in loss of information. Combining oversampling and undersampling techniques can be beneficial to balance the dataset while minimizing the loss of information.

Handling outliers
Outliers are data points that deviate significantly from the rest of the dataset. Outliers can skew statistical analyses and affect the performance of ML models. Techniques for handling outliers include trimming (removing extreme values), capping (replacing extreme values with a predefined threshold) or transforming the data to be more robust to outliers. When working with log data not all output is normal text, this can be due to numerous varied factors, either it is intentional or there can be some bug in the system. These occurrences can in this case be categorized as what are known as "Error outliers" [19] which are points in a dataset that is not of interest to the population. In this case, the population would be text.

2.1.2 Tokenization

When working with strings the data must first be tokenized before being inputted into a classifier. The process of tokenization is essential when machines need to understand and process human language. Tokenization involves breaking down unstructured text, which can be anything from sentences to entire documents, into smaller units called tokens [20]. These tokens could be individual words, subwords, or even characters, depending on the specific requirements of the task. For instance, when tokenizing a sentence, each word might become a token, or the words might be further segmented into chars. After tokenization, the tokens are converted into numerical representations through vocabulary building, which assigns a unique index to each token. To help with understanding how the tokenizers work, each of the different tokenizers will be accompanies by an example taken from Listing 1

"?1|20:52:35.395 [go] Task status: failed (147597 ms) (exit code: 1)" showing how this string is tokenized.

Word tokenization
Word tokenizer scans through the input text character by character, identifying patterns such as spaces, punctuation marks, and special characters that separate words or indicate word boundaries. During this process, it detects word boundaries using criteria like whitespace characters, punctuation marks, and language-specific rules. Special cases like contractions, hyphenated words, and abbreviations are also handled. Various word tokenizers treat special cases differently, but this thesis will utilize the NLTK Treebank word tokenizer [21]. As it processes the text, the tokenizer generates a sequence of tokens, each representing a single word or a meaningful unit of text such as a special character. As can be seen by Listing 2 where spaces are excluded from being tokenized, instead they only act as a separator.

```
{0: '(', 1: ')', 2: '1', 3: '147597', 4: '1|20:52:35.395', 5: ':',
6: '?', 7: 'Task', 8: '[', 9: ']', 10: 'code', 11: 'exit',
12: 'failed', 13: 'go', 14: 'ms', 15: 'status', 16: ''}
```

Listing 2: Tokenized mapping of words.

The final output array of tokenizing the input string would then look like in Listing 3.

```
[6, 4, 8, 13, 9, 7, 15, 5, 12, 0, 3, 14, 1, 0, 11, 10, 5, 2, 1]
```

Listing 3: Tokenized input with the word tokenizer.

Character tokenization
Character tokenization works by iterating through each character in the text, including letters, digits, punctuation marks, and whitespaces. For each character encountered, the tokenizer generates a token representing that character. It does not consider word boundaries or spaces; instead, it treats each character as a separate unit. The tokenizer continues this process until it reaches the end of the input text, producing a sequence of character tokens. For the example text provided above the character tokenization would produce the mapping shown in Listing 4.

```
{0: ' ', 1: '(', 2: ')', 3: '.', 4: '0', 5: '1', 6: '2', 7: '3',
8: '4', 9: '5', 10: '7', 11: '9', 12: ':', 13: '?', 14: 'T',
15: '[', 16: ']', 17: 'a', 18: 'c', 19: 'd', 20: 'e', 21: 'f',
22: 'g', 23: 'i', 24: 'k', 25: 'l', 26: 'm', 27: 'o', 28: 's',
29: 't', 30: 'u', 31: 'x', 32: '|', 33: ''}
```

Listing 4: Tokenized mapping of characters.

The final output array of tokenizing the input string would then look like in Listing 5.

```
[13, 5, 32, 6, 4, 12, 9, 6, 12, 7, 9, 3, 7, 11, 9, 0, 15, 22, 27, 16,
0, 14, 17, 28, 24, 0, 28, 29, 17, 29, 30, 28, 12, 0, 21, 17, 23, 25,
20, 19, 0, 1, 5, 8, 10, 9, 11, 10, 0, 26, 28, 2, 0, 1, 20, 31, 23, 29,
0, 18, 27, 19, 20, 12, 0, 5, 2]
```

Listing 5: Tokenized input with the characters tokenizer.

Byte-Pair-encoding tokenization

Byte Pair Encoding (BPE) tokenization begins with initializing a vocabulary containing all the characters or symbols present in the corpus. Next, the algorithm iterates over the corpus and identifies the most frequent pair of adjacent characters or character sequences. It merges the most frequent pair into a new symbol and updates the vocabulary accordingly. This process continues for a specified number of iterations or until a certain vocabulary size is reached. As the iterations progress, the algorithm gradually builds a vocabulary of subword units that represent frequently occurring character sequences in the corpus. During tokenization, the input text is segmented into subword units based on the vocabulary learned during training.

The tokenizer then replaces rare or out-of-vocabulary words with a combination of subword units that are present in the vocabulary. Byte Pair Encoding (BPE) tokenization is effective for handling rare words, morphologically rich languages, and out-of-vocabulary terms by breaking them down into smaller, more manageable subword units. Using ChatGPT4 tokenizer which has already been handed a full corpus, tokenizing the above message results in the following dictionary shown in Listing 6 for the above provided sentence.

```
{30: '?', 16: '1', 91: '|', 508: '20', 25: ':', 4103: '52',
1758: '35', 13: '.', 19498: '395', 510: ' [', 3427: 'go',
60: ']', 5546: ' Task', 2704: ' status', 4745: ' failed',
320: ' (', 10288: '147', 24574: '597', 10030: ' ms',
8: ')', 13966: 'exit', 2082: ' code', 220: ' '}
```

Listing 6: Tokenized mapping of BPE using ChatGPT4 corpus.

The final output array would then look like in Lisiting 7.

```
[30, 16, 91, 508, 25, 4103, 25, 1758, 13, 19498, 510, 3427,
60, 5546, 2704, 25, 4745, 320, 10288, 24574, 10030, 8, 320,
13966, 2082, 25, 220, 16, 8]
```

Listing 7: Tokenized input with the GPT4 tokenizer.

The dictionary when using GPT4 tokenizer can be compared to not already having a full corpus that would look like it does in Listing 8. Here the full corpus is the log data that is provided.

```
{1: '?1|20:52:35.395', 2: '[go]', 3: 'Task', 4: 'status:',
5: 'failed', 6: '(147597': 1, 'ms)', 7: '(exit', 8: 'code:',
9: '1)'}
```

Listing 8: Tokenized mapping of BPE using corpus from the provided text.

As the mapping in Listing 8 does not have the same context as ChatGPT4 the tokenizer tries to merge pairs of characters, often characters are only once beside each other in a word in this case.

LabelEncoder
The LabelEncoder is a predefined module within the Python package Scikit-learn, accessible through the pip library. It accepts input data in the form of an array and outputs corresponding tokens. Unlike other tokenizers utilized in this thesis, the LabelEncoder offers versatility by effectively serving as a substitute for various tokenization methods. For instance, the character tokenizer's functionality can be replicated by segmenting the log into an array of characters and feeding it into the LabelEncoder. The tokenized mapping of the string would therefore look like Listing 4 and then the output would be as shown in Listing 5. Without first segmenting the log into characters the tokenizer would instead output as is shown in Listing 9

```
{'?1|20:52:35.395 [go] Task status: failed (147597 ms) (exit code: 1)':
0}
```

Listing 9: Tokenized mapping of the entire input text.

The final output array of tokenizing the input string would then look like in Listing 10.

```
[0]
```

Listing 10: Tokenized input with the LabelEncoder.

The reason for this behavior with LabelEncoder is that it necessitates manual division of the text into smaller inputs by the user. With proper segmentation, it can perform the same function as other tokenizers. For instance, if the text is divided into two strings within an array beforehand, the resulting output array would resemble the depiction shown in Listing 11.

```
[0, 1]
```

Listing 11: Tokenized output from Labelencoder when two strings are sent in one array as input.

2.1.3 Training

Once the data is preprocessed and tokenized, it is divided into training and test sets. The training dataset is used to teach the classifier to recognize patterns [22]

and make predictions based on the input data. The test dataset is employed to fine-tune the classifier's hyperparameters and assess its performance during training, helping to prevent overfitting, where the classifier memorizes the training data but fails to generalize to new data.

Training an ML classifier involves feeding the preprocessed data into the classifier and iteratively adjusting its parameters to minimize the difference between the predicted outputs and the actual outputs. Other options besides hyperparameter tuning are to change how the preprocessed data is structured. This can be done by removing unwanted features or using several types of tokenization. This optimization process typically involves using algorithms such as gradient descent to update the classifier's hyperparameters based on the computed loss or error.

The algorithm starts at any random point with an initial set of parameters [23]. in the example provided in Figure 2.1 the start point is at the cost of 10000. At that point, the gradient of the terrain is calculated, which tells the algorithm the direction of the steepest slope. In ML, this gradient represents the direction of the steepest increase in the loss function, in the case of Figure 2.1 this is towards 0 where the start point is at 100. A loss function, in the context of ML and optimization algorithms, is a mathematical function that measures the discrepancy between the predicted values of a classifier and the actual ground truth values. It essentially quantifies how well the model is performing on a particular task. The goal of training a classifier is to minimize this loss function, thereby improving the classifier's ability to make accurate predictions.

Once the algorithm has found the direction of the steepest slope, it will try to take a small step in the opposite direction. This step size is called the learning rate, and it determines how far it is possible to move in each iteration and this is the same as each red dot in Figure 2.1. The current position is then updated to the new position that has been reached after taking the step downhill. This process is then repeated iteratively [23], recalculating the gradient, taking a step downhill, and updating the position until a point is reached where the gradient is close to zero or until a predefined number of iterations has been reached. In Figure 2.1 the number of iterations was reached before it could reach zero.

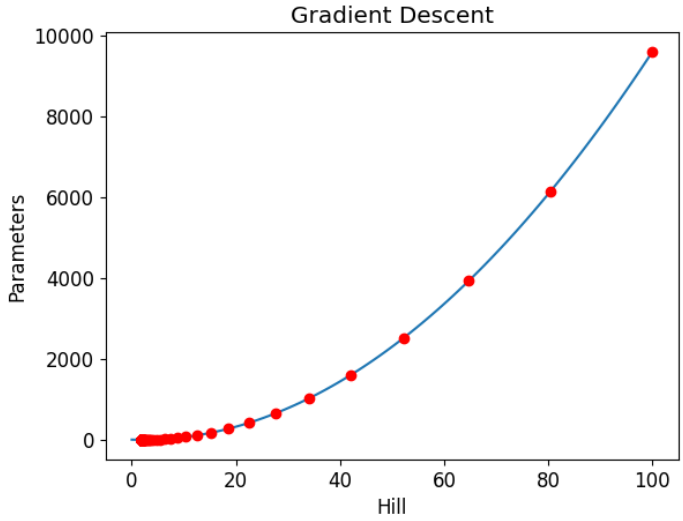


Figure 2.1: Example of how the gradient descent algorithm tries to reach zero.

The training process is iterative and computationally intensive, especially for complex models or large datasets. It often requires significant computational resources, such as powerful CPUs or GPUs, to expedite the training process. Additionally, techniques like parallel processing and distributed computing may be employed to accelerate training and handle large volumes of data efficiently. Having ample data necessitates having access to an abundant amount of memory since the model remains stored in memory throughout the training process.

Throughout the training process, monitoring and optimization are essential to ensure that the model converges to a satisfactory solution and generalizes well to unseen data. This may involve monitoring performance metrics, adjusting hyperparameters, and incorporating feedback from the test set to fine-tune the model's architecture and parameters.

2.1.4 Overfitting and Underfitting

When the classifier is training on previously unseen data overfitting and underfitting can occur. Overfitting and underfitting are fundamental concepts in ML that relate to how well a classifier learns from training data and generalizes to new, unseen data.

Overfitting
Overfitting occurs when an ML classifier learns the training data too well [24], including noise and random fluctuations in the data, to the extent that it negatively impacts the classifier's ability to generalize to new, unseen data. Essentially, the classifier memorizes the training data rather than learning the underlying patterns. As a result, an overfitted classifier performs very well on the training data but poorly on new, unseen data. Signs of overfitting include a high accuracy on the training dataset but a significantly lower accuracy on the validation or test datasets. Overfitting often happens when the classifier is too complex relative to the amount and

quality of the training data, or when the classifier is trained for too many iterations. To address overfitting many different techniques can be used, these are some of them:

Cross-validation: In certain scenarios, the ML classifier used can show very good accuracy on one part of a dataset but may not be generalized on other parts of the dataset [25]. Cross-validation involves partitioning the dataset into multiple subsets, known as folds. The classifier is trained on a portion of the data and validated on the remaining folds. This process is repeated multiple times, with each fold serving as both the training and test set in turn. Performance metrics obtained from each iteration are then averaged to provide a more robust estimate of the classifier's performance.

Training with More Data: Increasing the size of the training dataset can help the classifier to learn the underlying patterns better and reduce overfitting [24]. If obtaining more data is feasible, it is often one of the most effective ways to mitigate overfitting. This however comes with the tradeoff that more computing power will be needed as the ML classifier needs to fit the new data.

Underfitting

Underfitting, on the other hand, occurs when a ML classifier is too simple to capture the underlying structure of the data. In this case, the classifier fails to learn the patterns present in the training data and performs poorly on both the training and unseen data [24]. Underfitting can happen for assorted reasons, such as using a classifier that is too simple, not providing enough training data, or insufficient training time. Signs of underfitting include low accuracy on both the training and validation/test datasets. There are lots of different strategies to mitigate underfitting and the ones the thesis will cover are:

Increase Classifier Complexity: Underfitting happens when the classifier is too simple to capture the underlying patterns in the data. One way to try and solve this problem is by increasing the complexity of the model by adding more layers (only for neural networks), increasing the number of parameters, or using a more complex algorithm.

Feature Engineering: Feature engineering revolves around the creation, selection, and transformation of features from the original dataset to facilitate better classifier learning and prediction [26]. Effective feature engineering entails extracting relevant information, reducing dimensionality, and encoding domain knowledge into the feature space, thereby enabling classifiers to capture underlying patterns more effectively. One way of extracting features from text is the use of different tokenizers which will transform the text in diverse ways.

Hyperparameter Tuning: Hyperparameters are parameters that govern the behavior and performance of ML algorithms, distinct from classifier parameters that are learned from the training data [27]. Hyperparameter tuning involves systematically exploring and selecting optimal values for these parameters to enhance classifier

performance and mitigate issues such as overfitting and underfitting. In order to more easily find what parameters are most suited for the use case, several algorithms can be used to find the perfect fit. The two most popular algorithms are RandomSearchCV and GridSearchCV.

GridSearchCV embodies the concept of hyperparameter tuning by exploring a specified grid of hyperparameter values for a given ML algorithm. It systematically evaluates the performance of the classifier for all combinations of hyperparameters using cross-validation. On the other hand, RandomizedSearchCV operates on the premise of hyperparameter optimization by randomly sampling hyperparameter values from specified distributions or ranges. It systematically evaluates the performance of the classifier for each sampled configuration using cross-validation. The performance difference between them can be quite large meanwhile the difference in accuracy is not that big [27]. The search method of choice will therefore be RandomSearchCV when evaluating different jobs and how the parameters affect the result. GridSearchCV will however be used in order to get a baseline for the deviation between the two classifiers for this thesis.

2.1.5 Supervised learning

Supervised learning is a type of ML paradigm where the classifier learns from labeled data, meaning each input in the dataset is associated with a corresponding output or target variable [28]. The goal of supervised learning is to learn mapping from input variables to output variables based on the labeled training data provided. This is shown by the mathematical function 2.1.

$$f : x -> y \tag{2.1}$$

Where the data inputted and outputted into the function in 2.1 is formatted as follows in equation 2.2.

$$\{(x_1, y_1), (x_2, y_2)...(x_n, y_n)\} \tag{2.2}$$

In supervised learning, the algorithm learns from examples, where it is presented with input-output pairs and adjusts its internal parameters to minimize the difference between the predicted output and the actual output. In equation 2.2 the input is x and the corresponding output is y. For CI classification, y is the status of the finished job, either 1 for a successful job and 0 for an unsuccessful job. Parameter x is the input to the classifier, in the case of the CI job this can be the log data, files edited, commit author etc.

In classification tasks, the output variable is a categorical value or class label. The goal is to classify input data into predefined categories or classes. Examples of classification tasks include spam detection in emails, sentiment analysis in text data, and image classification.

2.1.6 Evaluation metrics

Researches usually resort to using commonly accepted performance metrics while evaluating the classifier [29]. Some common ones are accuracy, area under the

curve (AOC), area under the ROC (receiver operating characteristic) curve, and F-measure. Although these will be used in the thesis for evaluating the work of other studies, Matthews Correlation Coefficient (MCC) and Dynamic Time Warping (DTW) are used for evaluation of the results from this thesis.

The F1 score was not used in the thesis as it focuses solely on the positive class and does not take into account the true negatives. This can be a limitation in scenarios where the performance on the negative class is also important as in this thesis where the purpose is to be able to successfully predict CI builds that are about to fail. ROC was not used as it tells how well the model ranks positive instances relative to negative ones, but not about the actual decision boundaries. It is also not particularly useful for imbalanced classes, hence the decision to use MCC instead and also why AOC was selected out.

Dynamic Time Warping
Dynamic Time Warping (DTW) is a technique used in the field of time series analysis to measure the similarity between two sequences that may vary in time or speed. It is particularly useful when comparing sequences that may have temporal distortions, such as varying speeds, shifts, or nonlinear distortions. DTW finds an optimal alignment between the two sequences by warping the time axis, allowing for the comparison of corresponding points in the sequences, even if they occur at different times.

Matthews Correlation Coefficient
The Matthews Correlation Coefficient (MCC) is a metric used to evaluate the performance of classification algorithms, particularly in the context of binary classification tasks. It takes into account true positives, true negatives, false positives, and false negatives to provide a balanced measure of a classifier's performance, even in cases of class imbalance.

MCC ranges from -1 to +1, where different scores means the result can be interpreted as follows:

- A score of +1 indicates perfect prediction, where there are no false positives or false negatives.
- A score of -1 indicates perfect disagreement between prediction and observation.
- A score of 0 indicates random prediction.

2.2 Traditional machine learning

Traditional machine learning encompasses a collection of algorithms and methodologies employed to construct models capable of discerning patterns and generating predictions from data. These classifiers possess the ability to autonomously learn from data and make predictions or decisions without the need for explicit programming tailored to these tasks [30]. This section provides an overview of the inner workings of two classifiers, Random Forest (RF) and Gradient Boosting (GB).

2.2.1 Random forest classifier

Random Forest (RF) operates by constructing an ensemble of decision trees during the training phase. Each decision tree is built on a random subset of the training data, employing bootstrapping, or bagging techniques to introduce diversity among the trees [31]. Moreover, at each node of the decision tree, a random subset of features is considered for splitting, enhancing the robustness of the ensemble.

A decision tree is built by asking questions about distinctive features and at the start, all data is in one big box, which is the root node. Then questions are asked to split the data into smaller groups based on the answers. These questions are chosen to maximize the information gain or minimize impurity at each step.

As more questions are asked and data is split, branches are created that lead to more specific groups of data. Eventually, the tree ends up with smaller boxes, or leaf nodes, where no more questions are asked because the stopping point has been reached. In classification, each leaf node represents a class, and in regression, it represents a predicted value. When a prediction is made for a new data point, the algorithm starts at the root node and follows the branches based on the answers to the questions until a leaf node is reached. In classification, the majority class in that leaf node is the prediction, and in regression, it would be the average of the target values.

Decision trees are simple and interpretable. It can easily visualize the decision-making process, understanding which features are most important for making predictions. However, decision trees can also be prone to overfitting, especially if they grow too deep, capturing noise in the data instead of true patterns. In order to eliminate overfitting multiple trees are grouped into a forest.

During the prediction phase, the ensemble of decision trees collectively contributes to the final output. For classification tasks, RF employs a majority voting mechanism, while for regression tasks, it averages the outputs of individual trees. This aggregation strategy ensures robust and accurate predictions across diverse datasets.

The strengths of RF lie in its ability to mitigate overfitting [31], handle high-dimensional datasets, and provide insights into feature importance. However, it may exhibit computational overhead, particularly with large datasets, and may not perform optimally in the presence of noise or outliers. Additionally, the interpretability of RF classifiers may vary depending on the context and complexity of the data.

As noted by Nasir et al. [32] the RF algorithm excels in both classification and regression tasks, making it an outstanding choice for analyzing log data, especially textual logs, in this thesis. Its appeal lies in its capability to handle high-dimensional data efficiently while delivering excellent performance. Moreover, RF is adept at managing imbalanced datasets, a common scenario in CI job logs where failed logs contain information not typically present in successful CI job runs.

2.2.2 Gradient boosting

Gradient Boosting (GB) is a ML technique that is all about teamwork of different classifiers [33]. GB works by starting with a basic understanding of the problem, which is like having a simple initial classifier. This initial classifier might not be very accurate, but it gives a starting point. Then, it looks at where it is making mistakes. Next, it uses another simple classifier, like a decision tree, that is good at correcting mistakes made in the initial classifier. This other classifier focuses on the areas where the first classifier went wrong and helps improve the predictions [33]. The classifier gets adjusted based on the feedback from this new model, which means the classifier is gradually getting better at solving the problem. This process is then repeated, bringing in more classifiers one by one, each focusing on various aspects of the problem and helping refine the predictions further. With each iteration, it reduces the errors and gets closer to the correct solution

However, GB will try not to rely too much on one specific classifier, because the classifier could lead to worse predictions. So, instead a concept called "learning rate," which controls how much GB listens to each classifier's prediction [34]. This helps prevent over-reliance on any single classifier and keeps the predictions balanced and accurate. This process is then repeated until GB is satisfied with the predictions or until it has reached a predetermined level of accuracy. Finally, all predictions are combined, weighing them appropriately based on their quality of predictions, to arrive at the final prediction.

2.3 Deep learning

Deep learning (DL) is a subset of ML that focuses on using neural networks with multiple layers to learn representations of data [35]. Unlike traditional ML classifiers, which may require manual feature extraction, DL classifiers automatically learn hierarchical representations of data directly from the raw input.

At the core of DL are neural networks, which are computational models inspired by the structure and function of the human brain instead of the traditional zeroes and ones used in normal computing tasks [36]. These networks consist of interconnected layers of nodes (neurons), where each node performs a simple mathematical operation on its inputs and passes the result to the next layer. Neural networks typically consist of an input layer, one or more hidden layers, and an output layer [36]. Each layer is composed of multiple nodes, and connections between nodes have associated weights adjusted during the training process.

2.3.1 Layers

The input layer is the initial layer of a neural network that receives input data and passes it to the subsequent layers for processing. It consists of neurons that represent the input features or dimensions of the data. The number of neurons in the input layer is determined by the dimensionality of the input data. For example, if

you are working with images where each pixel is a feature, the number of neurons in the input layer would be equal to the total number of pixels in the image. The input layer does not perform any computation itself; its primary function is to pass the input data to the subsequent layers, which are typically hidden layers, and eventually to the output layer.

A hidden layer refers to a layer in a neural network that sits between the input layer and the output layer. The term "hidden" implies that these layers are not directly observable from the input or output data; they are intermediary layers where the neural network learns to represent features or patterns from the input data.

The output layer is the final layer of a neural network architecture. It is responsible for producing the desired outputs or predictions based on the input data and the learned parameters of the network. The structure and configuration of the output layer depend on the type of task the neural network is designed to solve.

During training, the dataset is divided into batches of a predefined size. Each batch contains a subset of the training data. DL processes data in batches and the classifier learn to adjust the weights of connections between nodes in order to minimize the difference between the predicted output and the actual output for a given input. This process, known as backpropagation [37], involves iteratively adjusting the weights using optimization algorithms such as stochastic gradient descent. Once all data has been iterated through one epochs is completed. After completing one epoch, the training process may continue with additional epochs if needed to further refine the classifier's performance. Each epoch provides an opportunity for the classifier to learn from the entire dataset and improve its performance gradually.

The depth of DL classifiers refers to the number of hidden layers in the neural network. Deeper networks have the capability to learn more complex data representations but also require more computational resources and can be more challenging to train.

2.3.2 Activation functions

An activation function is a mathematical operation applied to the output of each neuron in a neural network. It serves as a gate, determining whether the neuron should be activated or not, based on whether the neuron's input meets a certain threshold. Activation functions introduce non-linearity, enabling the network to learn complex patterns and relationships within the text data. Fully connected layers follow the convolutional layers, allowing the neural network to perform higher-level reasoning and decision-making based on the extracted features. These layers connect every neuron from one layer to every neuron in the next, facilitating a comprehensive analysis of the text.

ReLU
The ReLU activation function has previously been shown to improve DL neural

networks [38]. The function works by outputting the max value of a function or 0. The ReLU function is preferred over other activation functions like sigmoid and tanh because it helps mitigate the vanishing gradient problem [39], which can occur during the training of deep neural networks. Additionally, ReLU tends to be computationally more efficient compared to some other activation functions.

Sigmoid
The Sigmoid function maps any real-valued number to a value between 0 and 1 [40]. It has an S-shaped curve, with an output close to 0 for large negative inputs, close to 1 for large positive inputs, and approximately 0.5 at input value = 0. This property makes it useful for tasks when trying to model probabilities or create binary classifications, as it can squash the output of a linear function into the range [0, 1], thus providing a probability-like interpretation. Sigmoid functions are less favored now due to issues like vanishing gradients, where the gradient becomes extremely small for very large or very small inputs, hindering effective learning during backpropagation.

Tanh
The Tanh function takes any real number as input and outputs a value between -1 and 1. It is essentially a rescaled version of the Sigmoid function, stretching from -1 to 1 instead of 0 to 1. Tanh has many of the same characteristics as the Sigmoid function in that it has an S-shape and that the function asymptotically approaches -1 as the input value approaches negative infinity and approaches 1 as the input value approaches positive infinity. In neural networks, the Tanh function is often used as an activation function in hidden layers because it is zero-centered and tends to make learning easier compared to the Sigmoid function, which suffers from the vanishing gradient problem. The zero-centeredness helps in mitigating issues related to vanishing gradients, making optimization more efficient.

2.3.3 Convolution Neural Network

Convolutional Neural Networks (CNNs) are a specialized type of deep neural network architecture commonly applied in text-based classification tasks [41]. Unlike the traditional use of CNNs in image processing, CNNs for text analysis operate over one-dimensional sequences of words or characters instead of two dimensions as is the case with image processing.

In text based CNNs, the convolutional layers apply filters of varying sizes [42] to the input text, capturing local patterns and features. Each filter will try to capture different patterns in the input text to get a clearer picture when moving to subsequent layers. The output of applying a filter to the input data is called a feature map. Each filter produces one feature map, and the number of filters in a convolutional layer determines the depth of the output volume. The depth corresponds to the number of filters or feature maps.

Convolutional layers are employed to extract local features or patterns from text

sequences that it trains the classifier on. One-way filters are used on text to show how a sentence is structured; this can be done by checking what tokens (tokenized sentence) are often beside each other. With the sequence from Listing 3 the convolutional neural network would learn that there is a pattern of a non-predetermined number of words and then a space in between. This is what is typically called a word in normal English. The tokens from listing 3 inputted into a CNN can be seen in Figure 2.2.

In order to learn the pattern an operation called convolution operation is involved. It involves sliding a small filter (also known as a kernel or feature detector) over the input data and computing the dot product between the filter and the overlapping region of the input. In Figure 2.2 this is shown as the input layer to the bottom of the figure converges to the middle layer in the figure. In this case the kernel size of 2 means two values from the bottom layer converges to one value in the middle layer, as is shown by the first two values 5 and 8 both having lines drawn to the leftmost box in the middle layer. The result of this dot product is a single value in the output, referred to as the "feature map".

The value for sliding can be controlled by setting the stride parameter that determines the step size at which the filter moves across the input data. A stride of 1 means the filter moves one token at a time, while a stride of 2 means it moves two tokens at a time. Larger strides result in smaller output volumes. In the case of Figure 2.2 a stride value of 1 means that the leftmost numbers 5 and 8 is computed first, 8 and 6 are thereafter computed. With a stride of 2 the values 6 and 11 would instead be computed after 5 and 8. After computing the dot product between the filter and the input, a non-linear activation function is typically applied element-wise to the resulting values in the feature map.

After convolution, pooling layers are often used to reduce the spatial dimensions of the feature maps while retaining important information. One such pooling technique is Max pooling and divides the input to it into a grid of non-overlapping regions and for each region takes the maximum value. This helps ensure the network is more computationally efficient and helps reduce overfitting. In Figure 2.2 the last step in the figure is pooling where the three inputs from the middle layer become one. With max pooling the largest of the three inputs would be the result, while the other two inputs would be discarded.

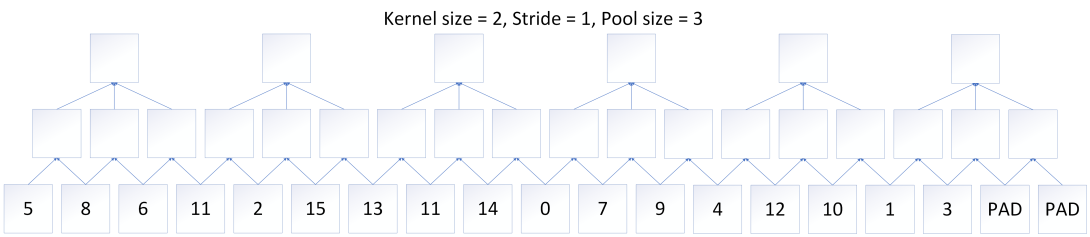


Figure 2.2: CNN network.

CNNs for text classification are trained using backpropagation, adjusting the weights

of the network to minimize the difference between predicted and actual class labels. Regularization techniques such as dropout and weight decay are often employed to prevent overfitting during training. Transfer learning can also be leveraged in text based CNNs, where pre-trained models trained on large text corpora are fine-tuned for specific classification tasks with smaller datasets. This approach enables the utilization of knowledge learned from the pre-trained models, enhancing performance and efficiency.

2.3.4 Long short-term memory

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) architecture. RNN is a type of neural network architecture tailored to process sequential data. The network maintains an internal state, or memory, enabling the network to capture dependencies between elements within a sequence, how this is done can be seen in Figure 2.3. One problem with RNN is the vanishing gradient problem [43] which occurs when the gradients used to update the weights of the network during the training process become extremely small as they are propagated backward through the network layers.

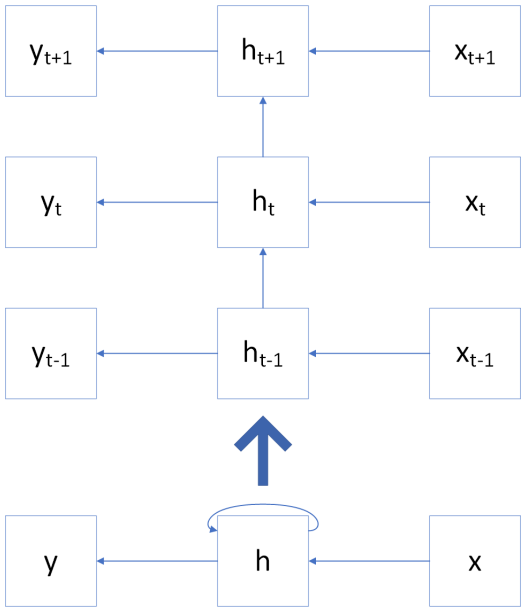


Figure 2.3: Basic RNN structure.

LTSM is specifically designed to address the issues of capturing long-term dependencies and avoiding the vanishing gradient problem that often arises in traditional RNNs [44]. At the core of an LSTM unit is the concept of a "cell state," which acts as the memory of the network. This cell state can maintain information over long sequences, enabling the network to retain context and dependencies over time. LSTM units have three main components: the input gate, the forget gate, and the output gate [44]. Each gate is essentially a set of mathematical operations driven by neural network layers, particularly sigmoid and tanh layers.

At each time step, the LSTM decides what information to discard from the cell state using the forget gate. This gate considers the previous cell state and the current input and outputs a value between 0 and 1 for each component of the cell state, where 0 means "completely forget" and 1 means "completely remember". Input Gate: Simultaneously, the input gate determines what added information to store in the cell state. It involves two stages, a sigmoid layer that decides which values to update and a tanh layer that creates a vector of new candidate values that could be added to the cell state. The forget gate and the input gate work together to update the cell state. The forget gate decides what information should be removed or retained, while the input gate decides what added information should be added. Finally, the output gate controls what information from the cell state should be output at the current time step. This gate filters the cell state through a sigmoid layer and produces the final output of the LSTM unit.

3
Related work

Several prior studies have explored the feasibility of anticipating the outcome of a CI job before its execution. These studies consider various parameters including modified files, edited lines of code, commit messages and more to predict the outcome. Beyond prediction on just CI jobs is other research on other types of log prediction that could possibly be used in order to predict CI jobs as well. Moreover others have not only investigated how to use predicitons in CI but in other areas related to improving developers workflow.

3.1 Traditional machine learning in CI

Saidani et al. suggests employing an evolutionary search algorithm [45], particularly focusing on the Non-dominated Sorting Genetic Algorithm (NSGA-II), for their research on cutting the CI job build time. Furthermore, they wanted to provide developers with support tools for setting up the rules characteristically for the developer CI system. This algorithm operates based on predefined rules established by developers manually adding them for various pipelines. These rules may encompass factors such as the number of changed files, the magnitude of changes, and so forth. Their optimal outcome yielded an AUC score of 0.78, accurately labeling data. Although the algorithm shows promise, this thesis will focus on making prediction in CI logs while the CI job is building, as such it would not be a feasible algorithm to use.

Xia et al. [46] looked at how it would be possible to predict if a CI build would succeed given the result from 126 different projects that contained almost 300.000 previous builds. They employed nine distinct classifiers, including decision tree, GB, logistic regression, multilayer perceptron, multinomial Naive Bayes, nearest centroid, nearest neighbors, RF, and ridge regression, to discern the most effective model. However, the findings were not encouraging, with none of the classifiers achieving an F1 score exceeding 0.6, indicating subpar performance according to the authors. The AUC score fluctuates more with scores as high as 0.8 and as low as 0.5. Notably, RF emerged with the highest F1 score in 0.4 of all projects among the classifiers tested. In terms of the AUC score, GB exhibited the most promise, leading in 75% of all projects assessed by the authors. Although the results were not promising as noted by Xia et al., the authors used the classifiers before any CI job had started. Therefore, the results could improve utilizing it while the CI job is running.

Several other studies has also looked at other ways traditional ML can be used in order to help with integration of code. One such study by E. Knauss et al. [47] looked at how it is possible to minimize the CI build time instead of predicting the build outcome. They researched how ML can be used to find out what test cases to select when running a test suite in CI. The goal of this was to reduce the time tests take in the CI build process.

The method used involved using a statistical model that connects changed fragments of code, also called code churns, as well as results of tests. The results show that using the approach could in the best case provide a 3.71 times speed up compared to running all test cases. In the worst case a 1.1 times speedup was achieved. This was in the context of one company, so this could vary between different companies but it still shows that there are other ways of improving the feedback time to developers without the use of CI build outcome prediction.

Another way to find failures before even starting a CI build is with code reviews. M. Staron et al. [48] state several problems with code reviews. Firstly that humans tends to focus on the wrong things when reviewing code, stating that this can be solved with filtering out commits that does not require manual review. Secondly, reviews can slow down the integration process in the case of long discussions were longer breaks removes the work flow. Also stating that configuration files are often part of the review, but can be hard for humans to read.

One possible way to solve this issue is with automating the code review workflow, although full automation is not desired as it can for example hinder knowledge transfer [48]. In order to find out what review comments are actually needed, the researchers classified what can be viewed as a positive or negative comment, correct classification in 72% of cases and incorrect classification of 28%.

These two papers show that not only prediction on the CI build outcome can help with the integration process but also other areas such as reducing CI build time through test case selection and making code reviews easier.

3.2 Deep learning in CI

Numerous studies have explored the application of DL in predicting CI build outcome. Saidani et al. [49]. conducted a study to evaluate the efficacy of deep learning using the same Travis CI dataset employed by Xia et al. [46]. Unlike previous research that employed various DL algorithms, this study focused solely on employing the LSTM classifier, complemented by a genetic algorithm for hyperparameter tuning. The findings revealed a more promising outcome, achieving an AUC score of 80% in certain scenarios, though results varied significantly, sometimes reaching as low as 50%.

The authors highlighted the considerable computational expense of the algorithm

but also acknowledged the potential for further optimization. The outcomes exhibited significant variation when applied to the Travis CI dataset; however, given its favorable findings, leveraging LSTM for near-real-time log analysis could prove advantageous.

When employing DL classifiers, they are often trained based on the structure of successful CI job logs compared to failed job logs. Failed CI job logs frequently contain anomalies when compared to successful CI jobs, rendering algorithms proficient in anomaly detection suitable for CI job log classification. Siyang et al. [50] explored three distinct DL clasifiers for anomaly detection within a Hadoop Distributed File System housing extensive logs exceeding 1GB. Evaluation of these algorithms yielded a Conventional Neural Network and Multilayer Perceptron achieving an impressive accuracy, F1 score, and AUC of 99%. In contrast, LSTM scored 95%.

While this thesis diverges from the studie's focus, it is noteworthy that CNN significantly outperformed LSTM. The authors emphasize CNN's suitability for classification tasks like the one in the study, as it addresses overfitting by capturing local semantic information rather than global data. Two key reasons underpin CNN's superiority compared to MLP for CI job runs.

Contextual learning in CNN: CNN models incorporate both horizontal embedding codes and longitudinal entries in logs through 2-dimensional convolution operations. This means that CNNs can learn the correlations between various log entries more effectively compared to MLPs. MLPs, on the other hand, have a simpler and faster training procedure but do not utilize contextual information as effectively as CNNs.

Relationships in log context with CNN: The CNN classifier can extract more relationships within the log context by leveraging multiple filters. Each line of parsed data belongs to a unique identifier group and is structured in a sorted order based on identifiers related to log events. This structured data presentation allows CNNs to extract meaningful relationships, such as the sequence of log events (e.g., "Job start" before "Job is running"). The CNN's convolutional layers can effectively extract these related features, leading to improved performance compared to MLPs.

3.3 Just-in time defect prediction

Kamei et al. [51] discusses three major problems with CI build outcome predictions. The first issue arises when predictions are made at a high level, leading developers to be uncertain about the causes of predicted outcomes. Secondly, if a particular file is predicted to contain a bug, it is essential to contact an expert responsible for that file. However, in some cases, more than 100 developers may be working on the same file, making it hard to know who to contact. Lastly, predictions are often made too late in the cycle, as developers aim to detect bugs as quickly as possible.

In contrast to the approaches prevalent at the time of publishing, Kamei et al. [51]

propose a new approach called just-in-time, which aims to prompt action at an earlier stage than previous methods. When employing the just-in-time paradigm, it can be triggered immediately upon a commit being made to a repository. Just-in-time refers to a principle or methodology that emphasizes delivering necessary resources or actions precisely when they are needed during the development process. By using just-in-time, the authors believe they address the three main problems. Firstly, the first issue is resolved by searching for each new detail in the commit instead of at the package/file level, resulting in smaller changes for developers to analyze. Secondly, each new commit has only one person attached to it, meaning that person is responsible for fixing it. Lastly, immediate feedback is provided as each commit is checked upon being committed to a repository.

To test the new approach, Kamei et al. [51] used a logistic regression algorithm where predictions above 0.5 were classified as defects, while others were classified as non-defects. The results of the study showed that only 20% of all bugs were found, leading to a poor result according to the authors. The authors further speculate that this may be due to checking if the change has introduced a defect or not, while others with better results check whole files and ignore which change caused the defect in the first place.

The problem addressed by Yang et al. [51] is whether it is possible to detect bugs in code written by developers. Their proposed solution, explored in their paper, involves utilizing DL with a novel approach named Deeper. The study aims to build upon the work previously conducted by [51]. Instead of relying on traditional machine learning methods like logistic regression, Yang et al. employed DL techniques, specifically utilizing the Deep Belief Network algorithm. Their new approach was then compared against the method proposed by Kamei et al. [51]. The results showed a significant improvement, with Yang et al.'s approach detecting 32.22% more bugs on average (51.04% versus 18.82%).

Recent advancements in just-in-time defect prediction have seen the adoption of novel algorithms aimed at enhancing prediction accuracy. Among these, the SZZ algorithm, pioneered by Sliwerski et al. [52], stands out as one of the most prevalent for identifying defect commits. Subsequent studies have explored variations of this original algorithm, such as those introduced by Kim et al. [53] and Da Costa et al. [54]. Yuanrui et al. [55] conducted a comprehensive analysis of these approaches to assess their susceptibility to mislabeling changes as either bug-fixing or non-bug-fixing. Their study revealed that both the Sliwerski et al. and Kim et al. methodologies exhibited high false positive and false negative rates. Specifically, the former recorded false positive rates ranging from 2% to 10% and false negative rates between 13% and 20%. Conversely, the approach by Kim et al. yielded false negative rates of 1% to 10% and false positive rates of 16% to 45%. Notably, Da Costa et al. achieved the most promising results, with false positive and false negative rates falling within the range of 1% to 6%.

Most defect prediction studies focus on identifying defects in commits before a CI

job runs, often without explicitly mentioning CI. However, they remain valuable because defect prediction can facilitate the resolution of issues before they reach the CI pipeline. This preemptive fixing could potentially render the subject of the thesis redundant. Nonetheless, existing defect-finding algorithms typically only detect around 50% of defects and are susceptible to significant levels of false positives and false negatives.

3.4 Bringing feedback to developers

With the predictions that are made by algorithms, it is also important that this information is displayed to the developer in a way that is easy for the developer to understand. Multiple different approaches have been suggested but all of them have focused on prediction algoritms that are done before the CI job has started.

One such implementation is a standalone application with a frontend for developers. Rosen et. al. [56] introduced Commit Guru, a web-based tool capable of analyzing commits within a repository. The tool tries to solve the problem of prediction tools not being used and the authors states that one significant reason for its limited adoption is the absence of tools that integrate state-of-the-art analytics and prediction techniques. It discerns which commits introduced bugs and which ones remedied them, alongside providing insights like total commit count and metrics indicating the level of risk associated with commits.

One limitation of the standalone web application approach is its lack of integration with git repositories [57]. Stating that there has been a growing concern among researchers regarding the necessity for software analytics to be both explainable and actionable. In order to remedy the shortcomings JitBot was created which is an acronym for a just-in-time defect prediction bot developed by Khanan et. al. [57]. The bot utilizes historical commits as training data to predict defects. Once a pull request is made, JitBot sends its predictions back to the developer, extracting necessary features from the data. It then comments on the pull request, explaining its predictions and suggesting potential mitigations. The authors illustrate two examples: in one, the bot identifies moderate risk due to developers' lack of experience and a high number of deleted lines. In another example of low risk, risks stem from a high number of modified directories and frequent changes to the files involved.

This thesis aims to address an important concern: the lack of evaluation regarding how developers would utilize such a system and whether they find it beneficial in their workflow. Additionally, the concept of "low experience" used in the paper [57] remains undefined, casting doubt on the effectiveness of their model.

Another approach to providing feedback to developers involves developing an extension to an Integrated Development Environment (IDE), as demonstrated by Kawalerowicz M. and Madeyski L. [58]. Their system, Jaskier, operates with a server responsible for training and predictions, along with a database for storage. Whenever a file is saved within the IDE, change statistics are transmitted to the

prediction server, which then sends the prediction back to the developer's IDE. The authors primarily focus on Visual Studio, which presents a graphical user interface indicating the likelihood of a file containing defects. For VS Code, this information is conveyed via the output console along with a corresponding percentage on fail probability.

This thesis seeks to analyze the different methods above to ascertain developers' preferences for receiving feedback from a prediction system. Moreover, as it also delves into predictions during CI build runtime, a topic largely unexplored in existing feedback, other potential solutions are also explored.

4
Research Design

The previous chapter outlined others' existing work and sets out a baseline for what result the thesis aims to achieve. This chapter follows the design science research (DSR) design paradigm [59]. It has more specifically followed the guidelines for conducting DSR in a master's thesis [60].

- **RQ1:** To what degree is it possible to provide feedback to the developer based on predicting a CI job build outcome?
  - **RQ1.1:** To what degree is it possible to predict the result of a CI job in just-in-time based on previous data from the same pipeline?
  - **RQ1.2:** What is the relationship between the computational power needed to predict the build outcome in just-in-time and the size of the job parameters?
  - **RQ1.3:** How can the feedback on the most probable failure cause from the just-in-time prediction best be displayed to the developer?
  - **RQ1.4:** What is the perceived usefulness of just-in-time build outcome predictions for software developers?
  - **RQ1.5:** At what point in the build process is it no longer useful to continue making predictions?
- **RQ2:** To what extent can the problems be solved by the potential solutions in (RQ1)?

Each research question posed defines one cycle, therefore the thesis features two cycles in total. The initial cycle delves into potential solutions for addressing these problems. Finally, the second cycle evaluates the efficacy of the solutions proposed in the second cycle in resolving the issues identified in the first cycle. The subsequent subsections delve deeper into the handling of each research question. Specifically, Section 4.1 addresses **RQ1**, while the subsequent section tackle **RQ2**.

4.1 Solution

In this thesis, the solution cycle was broken down into smaller iterations, each aimed at generating a distinct contribution to the artifact. These findings encompassed various evaluated combinations for the different classifiers, tokenizers, models as well as hyperparameters used. Also evaluated was their performance measured in terms of accuracy, MCC and required computational resources and the subsequent analysis of gathered information. The insights gleaned from analyzing the findings contributed to the subsequent iteration, thereby enhancing the iterative process.

Each iteration commenced with collaborative planning between the academic and industrial supervisors to outline the tasks to be undertaken. These plans were flexible, serving as rough guidelines rather than rigid directives. Adjustments were made as necessary, particularly based on the outcomes of earlier evaluations within the iteration, which could influence subsequent stages. The second part of each iteration consisted of building a prototype and evaluating different combinations for the artifact.

4.1.1 Machine learning model

The first part of the first cycle involved evaluation of different ML models that could best be used for CI build outcome prediction while the CI job is running. With ML there are a plethora of parameters that can affect the result and as such it is of utmost importance to evaluate different configurations for best performance. The different combinations comprised of various methods for improving the result such as parameter tuning, data handling and data balancing. The dataset for this study comprised of logs sourced from Zenseact's CI system, predominantly containing data related to the development of safety system software for automotive applications.

The system developed initially operated by employing a downloader capable of retrieving log files saved upon the completion of each real CI job. The downloaded metadata included the log file itself along with its corresponding status, denoting either a pass or fail outcome. This data served as input for the models utilized in the prediction system. Where the process was structured as follows by the bullet list and is connected to the flowchart in 4.1. Each number in the flowchart links to the explanation in the bullet list.

1. One year of job logs is downloaded from the specified job.
2. The downloaded CI job data is loaded, depending on the balancing used different data is loaded.
3. Data is preprocessed and unnecessary elements from the logs are removed if necessary. These can for example be timestamps.
4. The preprocessed data is tokenized with a tokenizer chosen for that build.
5. Lastly in the training phase, the tokenized data is handed over to a classifier which returns the trained classifier.
6. In the first step of simulating predicting a pipeline test data is loaded in iterations based on the number of lines to predict on for each prediction. This data follows the same steps as 3 and 4 for it to be handed over to the same classifier returned in step 5.
7. In this case the data returned will be a 0 if the classifier predicts the job will fail, otherwise it returns 1 if it predicts the job will succeed.
8. The predictions is evaluated against the real outcome of the job.

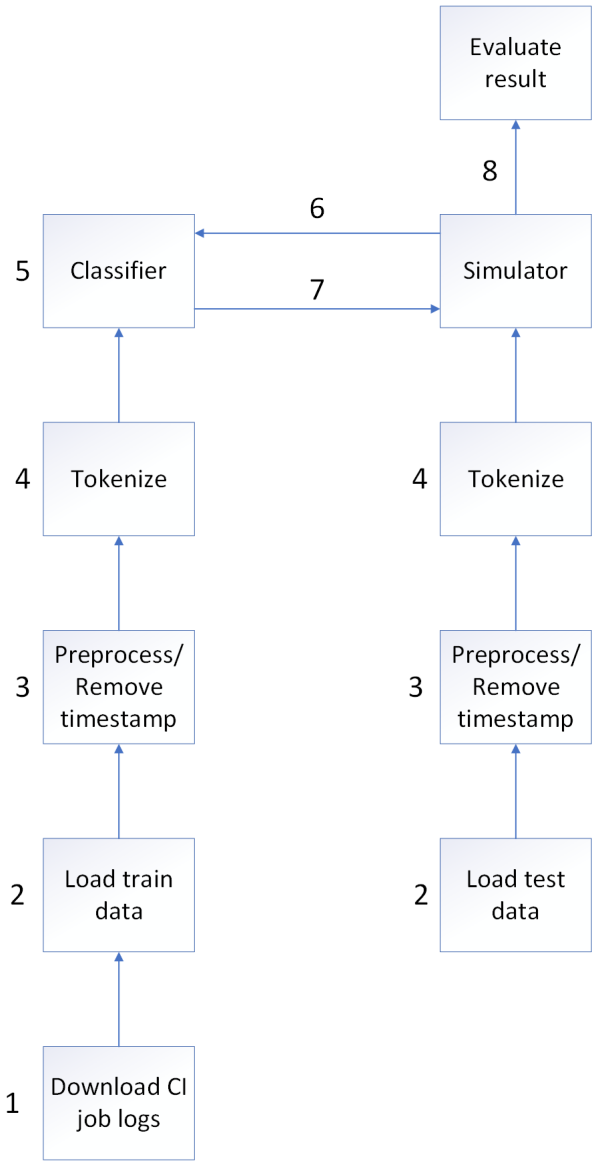


Figure 4.1: Flowchart of what is explained in the bullet list.

The concluding phase of each iteration involved analyzing the artifact and determining the course of action for the subsequent iteration, a process facilitated by collaboration with both industrial and academic supervisors. Each new finding served as a means to identify shortcomings and areas for improvement. Subsequently, planning for the next iteration commenced, with the primary aim being to address the identified shortcomings. Given the time-intensive nature of evaluating each combination, it was imperative to prioritize those combinations likely to yield meaningful data. Consequently, numerous combinations had to be discarded, resulting in a selection of only a few combinations to carry forward to the next iteration.

4.1.2 Interface for developers

Numerous methods exist for CI systems to deliver feedback to developers, and to determine the most effective ways according to developer preference, multiple de-

signs and approaches were compared for providing feedback from predictions. Two distinct approaches were employed in selecting which solutions to present to developers. The first involved an examination of the current feedback mechanisms utilized at Zenseact within their CI system. The second approach leveraged prior studies and the methodologies employed therein.

The adoption of these two approaches resulted in the creation of several mockups available in the interview guide found in the Appendix A, which were subsequently evaluated through interviews with developers at Zenseact. This process aimed to ascertain whether a consensus exists regarding the preferred approach or if developers hold differing opinions on the matter.

The selection of interviewees adhered to purposeful sampling, involving the identification of participants with expertise or experience in the field of developing software as well as those who were available and willing to engage, with an exemption to CI developers. Each interview spanned approximately 30 to 50 minutes, during which consent for data collection was obtained, and measures to maintain confidentiality were assured. Originally, the interviews were intended to exclusively involve developers working on the primary product, specifically self-driving technology. But in the end one scrum master and product owner was added in order to gain a broader view as they knew how the teams operate. All participants with their prior experience can be found in Table 4.1.

Table 4.1: Performance impact on training with different classifiers, models and tokenizers.

Name	Role	Years at Zenseact	Years as CI developer
Interviewee A	Developer	1.5	No
Interviewee B	Developer	7	No
Interviewee C	Developer	3.5	0.5
Interviewee D	Scrum master	3	2
Interviewee E	Developer	2	No
Interviewee F	Product owner	7	0.5

4.2 Evaluation

to evaluate the data gathered the thesis has employed a combination of a qualitative and quantitative evaluation approach. Research questions 1.1 and 1.2 have been analysed with the help of computational experiments and a quantitative evaluation approach. In order to answer questions RQ1.3, RQ 1.4 and RQ 1.5 a qualitative approach has been implored using the interviews as the base. RQ2 has subsequently been examined both using a qualitative and quantitative evaluation approach.

4.2.1 Computational experiments

During the analysis phase of each iteration for building the machine learning model hypothesis testing has been used. In some of the iterations time series analysis has also been used when involved with comparing the results from simulating a CI build.

The use of hypothesis testing begun with sampling, where a subset of CI jobs was selected from a larger population. The goal of sampling was to obtain a representative sample that accurately reflects the characteristics of the population, for this random sampling was used. An hypothesis was then built based on the result from the iteration and this was then either accepted or rejected in upcoming iterations. It involved formulating a null hypothesis (H0), which represents the status quo or no effect, and an alternative hypothesis (H1), which represents the claim to be tested. Hypothesis testing was the used to evaluate the evidence provided by the data in the iteration and to determine whether to reject the null hypothesis in favor of the alternative hypothesis, based on the observed results from the iteration.

In order to be able to determine whenever to accept or reject the null hypothesis multiple approaches has been used. The first is the use of multiple combinations for each job tested, meaning that not only one scenario is checked. Furthermore the use of cross-validation increases the likelihood that random sampling does not affect the final result. For the result of each combination tested an average of two different random samplings has been used.

Most of the data gathered are in a time series format and as such Time Series Analysis has been employed in these situations. Time series helps to spot trends in how predictions evolve as the CI build nears completion, this both involved trend analysis which aims to identify and model any long-term upward or downward movements in the data. Another important observation is Stationarity in which means that the statistical properties of the data do not change over time. For the Time series analysis DTW was used. By utilizing DTW it was possible to determine how much the different combinations evaluated differed between each other.

4.2.2 Interview

Before the interview took place two preliminary interviews was held with the supervisors in order to best determine the content of the real interview. The interviews was held either online through teams or on-site depending on the preference of the interviewee. The interview was semi-structured in that the questions was predetermined but the interview allowed the interviewee to come with solutions not mentioned in the interview guide. If this was the case additional questions about the solution was asked in order to clarify what the goal is that the interviewee is seeking.

In order to analyze the interview results, thematic analysis emerged as the preferred method for dissecting qualitative data. Widely acknowledged in the research realm, it stands out as one of the most prevalent approaches. This method places a strong emphasis on uncovering, examining, and interpreting themes or patterns

inherent in qualitative data. Notably, its procedural steps provide a versatile framework for data analysis. While thematic analysis shares several characteristics with other qualitative analysis techniques, its effectiveness remains undisputed.

The thematic analysis was divided into two separate sections as the interviews had two topics discussed. Firstly an analysis was done on the first part of the interview which discussed how feedback could best be sent to developers. Secondly, an analysis was done on the part trying to gather how good a prediction algorithm needs to be in order to be useful. Each of them followed the same procedure, with the first step involved writing transcripts of all interviews. Then each question posed got a label attached according to how the developer felt about the approach. The last step of the thematic analysis was to identify patterns in the interviews with the help of the labels created.

5
Artifact

This chapter introduces the final two artifacts and explores their practical applications. The first artifact is a feedback system, and the section outlines all the steps required to implement this system into a production environment. A flow diagram illustrates the integration process, accompanied by a detailed explanation of each step. The second artifact, a prediction system for real-time training and prediction, is presented in the final section. This system is also depicted with a diagram and includes a comprehensive guide to its implementation.

5.1 Feedback flow

The feedback system has three components that are vital to it operating, all can be seen in the Figure 5.1. The first component is the prediction system. The second component is the CI system that needs to send prediction requests to the prediction system. Finally is the notification system that is dependent on a service which can send notifications to e.g. email, Slack etc.

Before any predictions can be made a classifier has to be trained. The first step is for the data to be downloaded, this is handled on a set time interval set up by the developers. Once the data has been downloaded it is in step 1 it is used for training a classifier. In order to ensure the best possible results the classifier will automatically train multiple classifiers with different hyperparameters and use against a test sample. Once it has found the best combination, this will be saved for later use in step 2. All parameters used are configurable by the developers. When the right classifier is saved the training process is done with step 3.

The second step involves the CI system. All begins with the developer uploading a commit to a version control system such as Gerrit in step 4. Once the CI system is notified that a new commit is available it downloads it, and starts the build process in steps 5 and 6 respectively. When the log for the build has either reached a certain point in time or a threshold of lines written to the log are met the data is sent to a prediction system in step 8. To be able to make predictions the CI system has to be set up in such a way that it is able to detect when new log data has been written and send this log stream to the prediction system. In this case not only log data needs to be sent but also all metadata available, as this is crucial for choosing the right classifier.

Lastly, the data sent from the CI system is handled by the prediction system, which analyzes the metadata before deciding on the appropriate tokenizer, classifier and hyperparameters etc. to use. Once the data is processed and a prediction is made, if the predictor foresees that the build will succeed, step 8 concludes the process and no further action is taken. However, if a build failure is predicted to happen, in step 9 a notification service is used to inform the user with the necessary information. This is in the form of a message to were the predictions can be found. An example of this can be found in the interview guide in Figure A.4. This information is displayed to the user on what has caused the predicted build outcome to be a failure. A simple example of this can be found in Figure A.5.

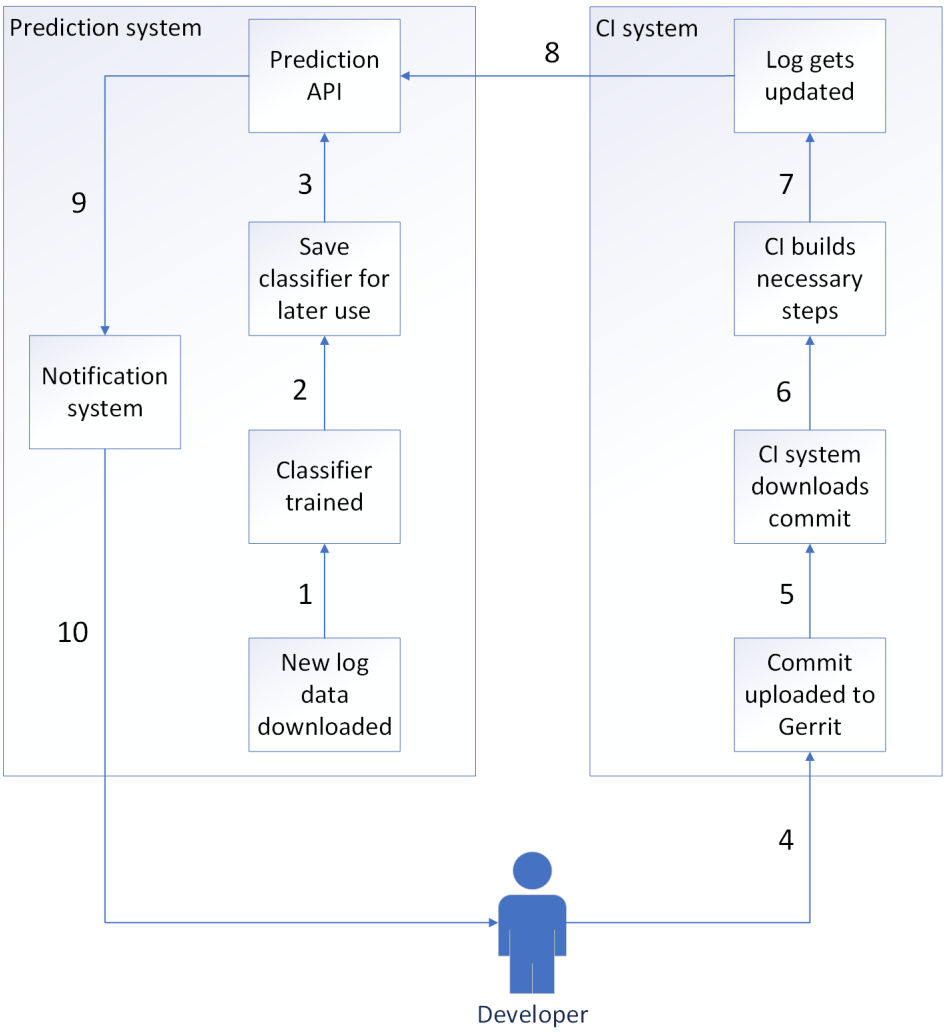


Figure 5.1: Flowchart of feedback loop.

In order to set up such a system into an already existing CI system, the CI system would need to be reprogrammed to be able to send metadata once a certain threshold has been reached. This would involve either a certain time of number of lines printed. For CI systems which does not have this capability, this would instead be handled by the prediction API polling the CI system for updates. This would then have to be based on a time interval. What has been described is step 8 in the feedback

flow in Figure 5.1. The prediction system would therefore be a separate system not connected in any way to the CI system, this is because the CI system would need to be independent on failures in the prediction system.

5.2 Prediction system

The prediction system presented in this thesis comprises of steps 1, 2 and 3 in the feedback flow Figure 5.1. The best way found to implement the system is to automate it so minimal human interaction is needed. The flow of how training is done is shown in Figure 5.2, which contains a more informational view compared to the overview in Figure 5.1.

At the center of the prediction system is the task distributor, this distributes tasks that needs to be done, there is a certain flow that needs to be followed namely: download data, train a classifier on the data, test the classifier against test data and evaluate the different classifiers and pick the best to be used by the Predictor API.

The first task is to download the necessary data for training. The downloaded data includes the log and the job status, which can be either successful or unsuccessful. The input received from the task distributor is the name of the job to download. Additionally, metadata for the job is downloaded, which can include information about which files were edited in the commit tested, what lines were changed, and what caused the job to fail. This downloaded data is then serialized and stored on disk in two separate folders: one for testing and one for training. Once the data is downloaded, the download process is terminated.

In step 3, the task distributor then launches a new instance of the training sequence, where a classifier is trained. The input for this training includes the combination to be tested, the job to train on, the model that should be used, the classifier to use, and the tokenizer. There are also optional parameters specific to the classifier. For traditional ML classifiers such as RF or GB, possible hyperparameters include the number of estimators, max depth, and min samples leaf, among others. For DL this includes epoch count, number of layers as well as the kernel size etc. The training is performed using the specified parameters, resulting in a classifier that is stored on disk in step 4, to be used by the tester and the prediction API. This training process is repeated multiple times by the task distributor; for each combination to be evaluated, a new training instance is called with the new parameters.

For each trained combination, the task distributor will call the tester with the exact same parameters to inform it which classifier to import from the disk. The tester will test all combinations against the test sample downloaded in step 1. Both the test sample and the classifier are loaded from the disk, and once a result is obtained, it is stored on the disk. The only requirement for the tester is that the training phase for the combination is done. As such it is possible that the Task distributor will distribute the tasks so that the RAM, CPU and GPU is utilized as much as possible. Therefore there might be one task requiring the GPU while the CPU is

used for other training tasks.

Lastly, the evaluator will assess all results obtained in step 6 and choose the best combination of classifier, model, tokenizer, and hyperparameters for a certain job. This classifier will then be sent to the Predictor API to make predictions from running pipelines. This is the same as can be seen in the feedback loop Figure 5.1 in step 3. The evaluation is based on parameters set by the developers; in this thesis, accuracy is measured from the end of the CI job setup until the end of the last unsuccessful job. However, these parameters can be adjusted to incorporate MCC or another metric, or if the developer prefers to make predictions until only 15 minutes are left of the job. The evaluator cannot run until all combinations have been processed through steps 3 to 6. While it will pick the best available combination, it may not have tested all possible combinations, which might affect the results.

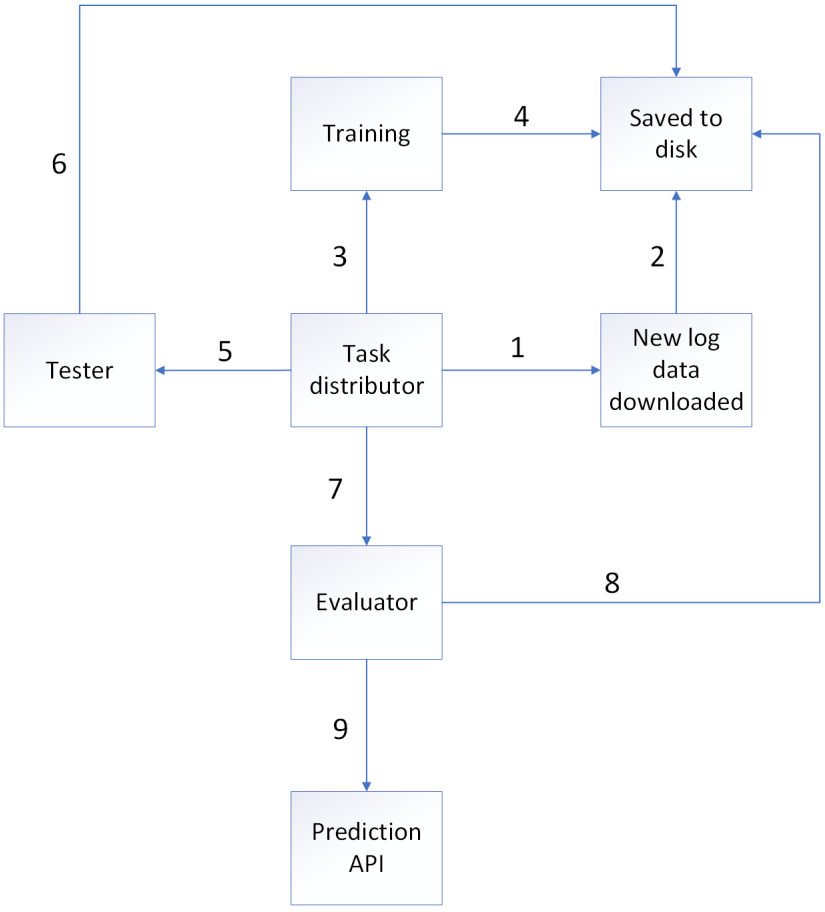


Figure 5.2: Flowchart of how the predictor functions.

6
Findings

This chapter presents the result and analysis of each iteration. First, an overview of what has happened in the iteration is presented, thereafter the result from that iteration is presented. Lastly, an analysis of the results is performed which lays the groundwork for the next iteration. All results have been gathered on a server running an Intel Xeon Gold 6248R with Ubuntu 22.04 as the operating system. For the DL testing an Nvidia GRID T4-16Q has been used to gather the result, featuring the full 16GB of VRAM. All models have been run with an 80/20 split meaning that 80% of the data has been used for training while 20% has been used for validating and testing to verify the model.

6.1 First iteration

The first iteration primarily consisted of setting up the development environment and downloading the necessary data for running ML classifiers. All jobs in the CI system follow the same structure as is seen in Listing 12. Listing 13 subsequently provides an example of how such a log looks like where the output type is "&1" indicating it is of the log level info written to standard output. Furthermore, the time written is at 20 hours 10 minutes and 29 seconds, where it also reports milliseconds after that. Lastly, the message written is that the CI system is currently waiting for an executor to run the tests.

```
[output_type] | [timestamp] [message]
```

Listing 12: The structure of a line in the job logs followed by an example.

```
&1|20:10:29.720 Waiting, In the quiet period. Expires in 3.8 sec
&1|20:10:29.720 Waiting, Finished waiting
&1|20:10:29.721 Waiting, Executor slot already in use
```

Listing 13: Example of three lines taken from a log in job 1.

The output_type in Listing 13 indicates the type of output logged by the system. Types of logging can be either standard output or error output but also other types like the ones used in Python. These can be INFO, DEBUG, CRITICAL, etc., but the CI system may also have specific output types tailored to certain messages, such as "Current job status: passed".

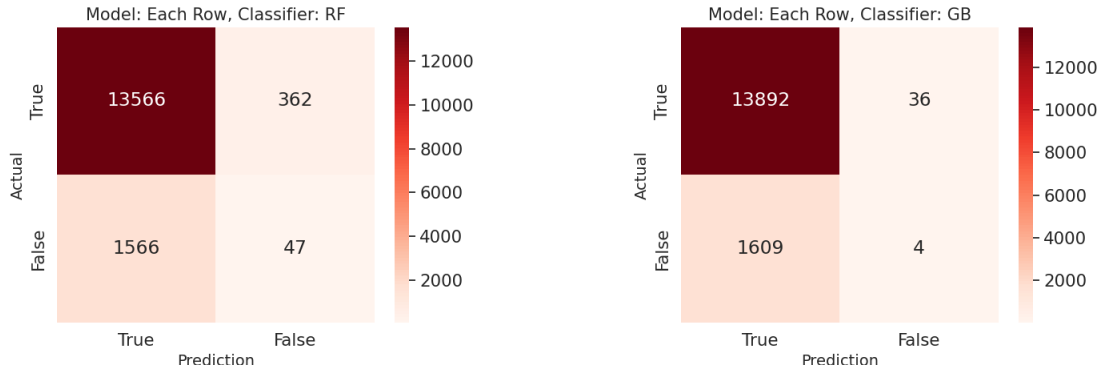
Job 1 encompassed 174 logs, comprising 154 instances of successful CI job logs and 20 instances of unsuccessful CI job logs, with a cumulative data size of 16.4 MB. Throughout the analysis, two distinct classifiers, RF and GB, were employed and juxtaposed across three different models. Throughout the first iteration, the Scikit-learn module LabelEncoder was exclusively utilized as a tokenizer, without the inclusion of any alternative tokenization methods. The comprehensive list of combinations evaluated during iteration 1 is provided in Table 6.1.

Table 6.1: Combinations evaluated in the first iteration.

Jobs:	Job 1
Classifiers:	RF
	GB
Tokenizers:	LabelEncoder
Models:	Whole Log
	Logtime
	Each Row

6.1.1 Result

The first model for the project is called “Each Row” which takes a log and splits it into multiple lines. Each line is further divided into three features in the same way the log is divided, shown in Listing 12. The first feature represents the log’s output type, the second denotes the timestamp indicating the time the log line was added to the log, and the third feature is the log message. Each feature is then tokenized using LabelEncoder. In Figure 6.1a depicts how accurate RF was when predicting on log files from job 1 using the Each Row model. The model in conjunction with RF scored an accuracy of 88%. Similarly, Figure 6.1b shows the confusion matrix metrics for the GB classifier. The GB classifier with the Each Row model got an accuracy of 89%. Each number in Figure 6.1 represents how many rows that have been assigned to a certain class, with correctness indicated if the class is accurate. However, it is essential to note that both models essentially predict "True" for most cases. Due to the significant class imbalance, this tendency towards predicting the majority class results in relatively high accuracy. Subsequently, Table 6.2 then presents the accuracy metrics from this same run. In the table it can be seen that the number of samples are identical indicated by the "support" column, but for all other metrics the two models varied were the RF classifier performed better in all metrics related to prediction of class 0, denoting a unsuccessful job. On the other hand, GB performed better in all metrics related to prediction on successful jobs, shown as class 1.



(a) **(b)**

Figure 6.1: Confusion matrix comparison of RF and GB on Each Row model.

Table 6.2: Accuracy metrics when running model Each Row with the LabelEncoder tokenizer. RF metrics to the left and GB to the right.

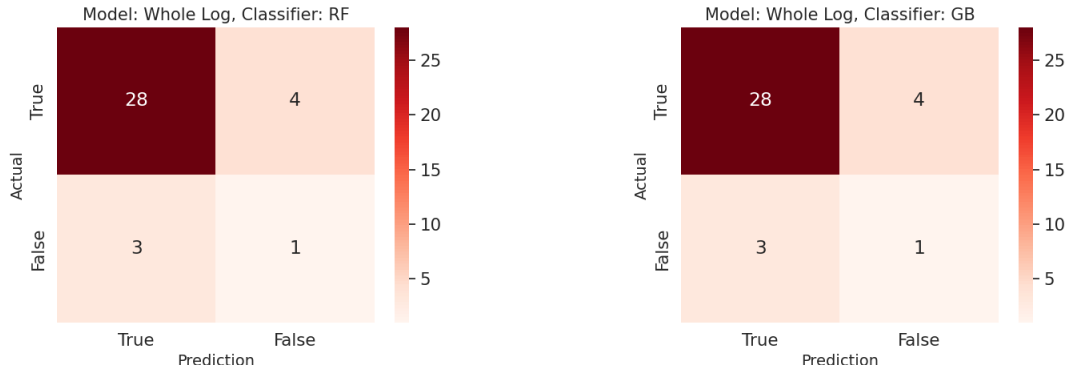
(a)

Class	Precision	Recall	F1-score	Support
0	0.11	0.03	0.05	1613
1	0.90	0.97	0.93	13928

(b)

Class	Precision	Recall	F1-score	Support
0	0.10	0.00	0.00	1613
1	0.90	1.00	0.94	13928

The second model named "Whole Log" model encompasses utilizing the entire log file as a single feature during training. This means that the entire log is tokenized into discrete elements and serves as the model's input. These tokens are encoded using the Scikit-learn tokenizer LabelEncoder. Essentially, this model trains on the entire log content, treating it as one cohesive unit rather than analyzing individual components separately. Figure 6.2 illustrates the classification of each log using RF in Figure 6.2a and GB in Figure 6.2b. Subsequently, Table 6.3 displays the accuracy metrics obtained from this. Contrary to the Each Row model, on the Whole Log model both classifiers got the exact same accuracy of 80% as well as the exact same accuracy metrics. Despite achieving high accuracy, the Whole Log model encountered a similar challenge as the Each Row model due to imbalanced data. This imbalance leads to inflated accuracy figures, primarily driven by high accuracy for class 1, while the accuracy for class 0 is deteriorated.



(a) **(b)**

Figure 6.2: Confusion matrix comparison of RF and GB on the Whole Log model.

Table 6.3: Accuracy metrics when running Whole Log model with the LabelEncoder tokenizer. RF metrics to the left and GB to the right.

(a)

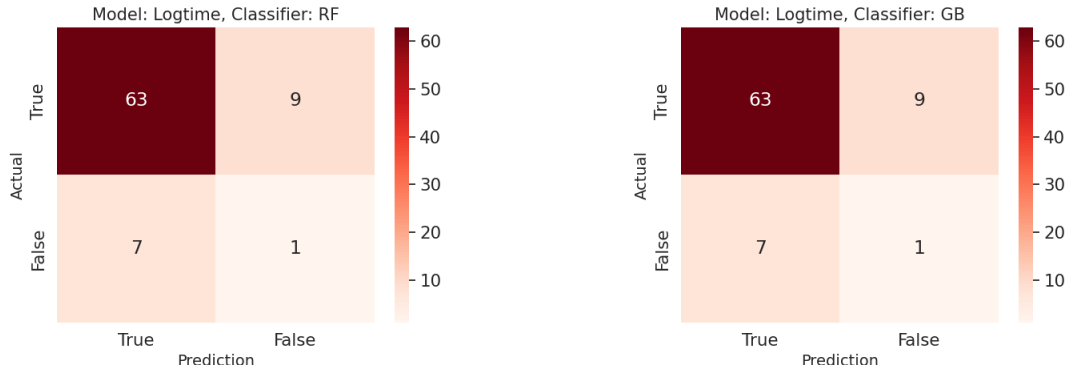
Class	Precision	Recall	F1-score	Support
0	0.20	0.25	0.22	4
1	0.90	0.88	0.89	32

(b)

Class	Precision	Recall	F1-score	Support
0	0.20	0.25	0.22	4
1	0.90	0.88	0.89	32

The final model employed in the first iteration, denoted as "Logtime", involved utilizing the timestamp as a mechanism for segmenting the log data into separate parts. This works by checking the timestamp for when a particular log line is written against a predetermined interval. If the time is below the interval, then the line gets appended to the feature that is going to be trained once all log data from the log has been added. If for example the time would split on each millisecond the log in Listing 13 would be two pieces in the dataset. The first piece would be the first two rows, while the second piece would be all three rows.

The application of the Logtime model to real log data from job 1 is depicted in Figure 6.3. Predictions were generated by partitioning each log into segments in the same way as the training data. In Figure 6.3, each numerical value represents a time-based split in the log, occurring at five-minute intervals. It shows no difference between the two classifiers. Table 6.4 displays the accuracy metrics obtained from this analysis. akin to the results of the Whole Log model the Logtime models also received the same accuracy metrics on both RF and GB. But on the other hand the accuracy of both classifiers was 81%, 1% point higher than Whole Log.



(a) **(b)**

Figure 6.3: Confusion matrix comparison of RF and GB on the Logtime model.

Table 6.4: Accuracy metrics when running Logtime model with the LabelEncoder tokenizer. RF metrics to the left and GB to the right.

(a)

Class	Precision	Recall	F1-score	Support
0	0.10	0.12	0.11	8
1	0.90	0.88	0.89	72

(b)

Class	Precision	Recall	F1-score	Support
0	0.10	0.12	0.11	8
1	0.90	0.88	0.89	72

6.1.2 Analysis

The first iteration focused mostly on being able to run the RF and GB classifiers on the full job log to ensure everything was working. These results should provide a baseline for what is achievable running these models on full job logs. It should be noted that the results should improve as there has not been any data balancing or hyperparameter evaluation done yet. When testing in this iteration the models had the most amount of data that is available for job 1. Most of the time the more data the better the prediction accuracy as the classifiers gets better at learning the underlying log structure.

All results gathered in the first iteration showed that both RF and GB accurately predicted the successful job but failed to classify the unsuccessful jobs. For the successful jobs the F1 score was always above 0.89 even reaching 0.94 with the Each Row model and GB classifier in Figure 6.1b. Prediction on unsuccessful jobs performed much worse were the F1 score reached 0% on the Each Row model with the GB classifier. There are several areas to explore for why this was the case, with the first being the use of LabelEncoder and sending in the entire log message instead of splitting up the log message. The Second noteworthy ascertainment is why all models successfully predicted successful jobs so well. The hypotheses going into iteration 2 is that successful jobs look the same in most cases and as such all successful jobs get the same token when the log is tokenized. Whereas there are also a great deal more successful jobs than unsuccessful jobs which could skew the result further requiring

data balancing in future iterations. For unsuccessful jobs, where anomalies are likely to occur, they all get different tokens but are mostly categorized as successful jobs. This is likely due to data balancing being in favor of successful jobs, as there is an over representation of successful jobs. To get a better look how balancing affect the result MCC will be used as a metric in conjunction with keeping the accuracy metric.

Tokenizers:
The key insights gleaned from the initial iteration underscore the importance of employing multiple tokenizers, including those commonly utilized in text classification. Iteration 2 aims to validate the hypothesis that the method of using LabelEncoder yielded inferior results compared to other tokenization techniques. Consequently, three tokenizers were selected for the second iteration:

The first tokenizer is a char tokenizer. As most of the successful jobs contain almost the same information, by making the ML models predict based on the occurrence of characters it could be possible to get a more accurate prediction on when a job has failed. A prediction based on the number of characters would essentially mean that the ML classifier learns that successful jobs generally exhibit consistent patterns, allowing it to identify anomalies, such as a unsuccessful job.

The second tokenizer, a word tokenizer built on Nltks TreebankWordTokenizer was chosen for the same reason as the char tokenizer. Often successful jobs will contain the same words, so it could be possible to predict their occurrence. Nltk tokenizer is an already established tokenizer for use with log data. With words, it should theoretically also reduce the memory needed to train the data as it takes more data to store the chars than a word.

The last tokenizer to be used is the one used in ChatGPT-4 (BPE): A lot of progress has been made in the last couple of years of perfecting byte-pair-encoding tokenizers and as such there is not much current research on how the latest tokenizer can handle a scenario which the thesis covers.

One concern is that solely predicting the occurrence of characters or words may pose risks, as the structure of logs can vary from one day to another. Therefore, leveraging logs spanning from the past year to the present can facilitate capturing the evolutionary trends in job log structures over time, enhancing the adaptability of the models.

Timestamp:
The Each Row model used timestamp as a feature, the entire timestamp was tokenized using one token with LabelEncoder. With iteration 1's approach, the only scenario where two timestamps would share the same token is if the timestamps themselves are identical. Since the timestamps go down to milliseconds, with an example shown in Listing 13, it means that almost all timestamps would receive different tokens in this case. With a word tokenizer, it would separate the different numbers as they all have a separator between them and as such should get a better

prediction on the timestamps. Although the prediction should theoretically improve, another concern is that the timestamp for each log should be removed. This is both to ensure the results are more generalizable and because the timestamp will not help in generating an accurate result. In a CI system, jobs are often offloaded from the main CI system to clusters of machines where the job is executed. As is also the case in the CI system used for this thesis. This can skew the results because the ML algorithms prioritize the time taken rather than the content of the CI job logs. In order to confirm this, in iteration 2 tests will be made on both timestamps kept and removed.

Successful and Unsuccessful dataset differences:
Figures 6.1, 6.2 and 6.3 indicate that the high accuracy is largely contributed by the significant number of successful builds while almost all the unsuccessful builds get the wrong prediction. In forthcoming iterations, testing is conducted separately on successful and unsuccessful jobs to compare the accuracy of predictions over time for each class of the job.

Models:
To accommodate additional combinations, including new tokenizers, a trade-off was necessary to maintain the time frame for iteration 2. Consequently, the decision was made to eliminate Each Row prediction model. The decision for removal was based on that it would not be possible to add information from previous rows of data in the log to the model. Each Row in the log would therefore be its separate unit with no context of the previous data, leaving it susceptible to underfitting. Despite its exclusion, iteration 2 still involves evaluating 72 different combinations, where all combinations are shown in Table 6.5.

6.2 Second iteration

The second iteration aimed to establish a simulator and use it to assess the predictive performance of the various combinations listed in Table 6.5. Additionally, the iteration involved testing multiple tokenizers to compare their performance. It was crucial to assess the predictive capabilities of the models and classifiers over time using the simulator, considering both successful and unsuccessful jobs, to proceed in future iterations effectively.

Table 6.5: Combinations evaluated in the second iteration.

Jobs:	Job 1
Classifiers:	RF
	GB
Tokenizers:	Char
	GPT4
	Word
Models:	Whole Log
	Logtime
Dataset used:	All jobs
	Only unsuccessful jobs
	Only successful jobs
Timestamp:	Keep
	Remove

One key point of the thesis is to see how much time can be saved at a certain point. Hence, Figure 6.4 illustrates the utilization of all test samples from job 1 and their respective duration's after surpassing certain line numbers. The graph delineates a considerable range in job completion times, spanning from approximately 500 seconds to 3700 seconds. Notably, some jobs conclude around the 350-line mark, while one extends to about 710 lines. This variability stems from two primary factors: firstly, the computational resources handling the job may be occupied with other tasks, leading to waiting periods. This is the reason the time taken sharply rises from one point to the next. Secondly, the job's configuration dictates selective execution, potentially altering the required configurations based on the task at hand, which is what causes the discrepancy in the number of lines for each sample.

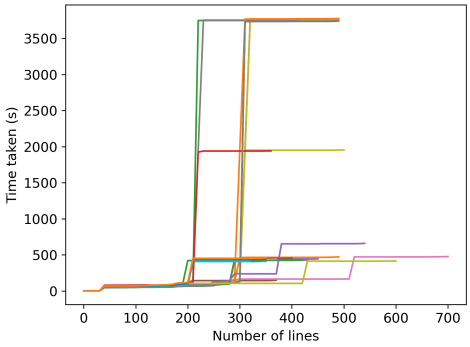
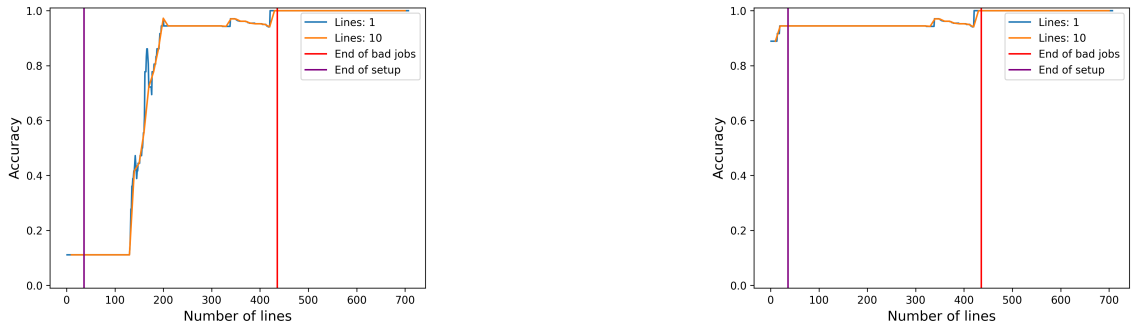


Figure 6.4: Time and line progression for each test sample in job 1.

6.2.1 Result

The finalized simulator comprises a program capable of reading a log and iteratively segments it into multiple parts. Each part is then predicted on, and the returned result is an array of predictions. This process iterates over predetermined sets of lines within the log until all log data has been predicted. Subsequently, the procedure is repeated for all logs, and the resultant predictions are aggregated to calculate the MCC score, the accuracy or the DTW distance. The graphical representation in almost all figures featured in this chapter illustrates the correlation between the number of lines processed and the corresponding metric used.

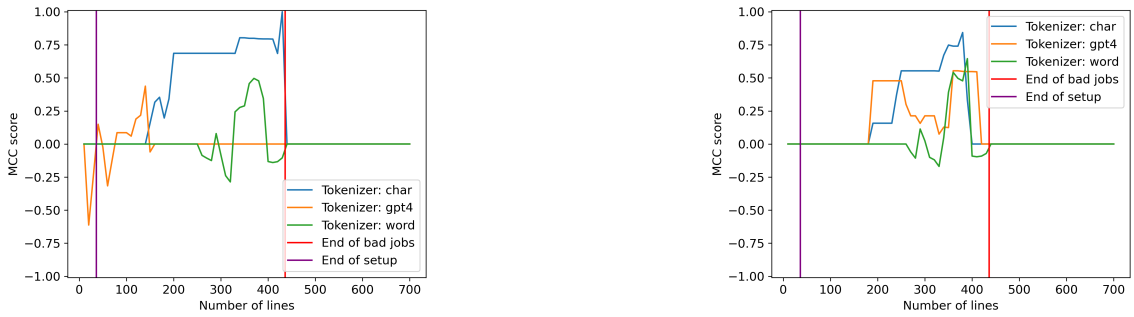
Each graph in Figures 6.6 and 6.7 illustrates the relationship between the number of predicted lines (x-axis) and the MCC score of predictions (y-axis). Consequently each graph in Figures B.1 and B.2 shows the relationship between the number of predicted lines (x-axis) and the accuracy of predictions (y-axis). To manage computational resources efficiently, the number of lines predicted increases in increments of 10. what this means is that the first predictions in made on the first 10 rows of log data, the next prediction is then made on 20 lines of log data and so on until the last prediction where all log data is used for a prediction. Increments of 10 has been chosen as incrementing on each line individually would incur significant computational overhead without substantial benefit due to minimal variability in predictions from line to line, shown in Figure 6.5. Each graph's caption is labeled with information regarding the classifier, tokenizer, model used, and whether timestamps were removed. Additionally, a purple line denotes the completion of the job setup, a step common to both failed and successful jobs (excluding those that fail during setup). A red line indicates the length of the longest failed job.



(a) Model: Whole. **(b)** Model: Logtime.

Figure 6.5: Comparison of predicting on each line compared to on only every tenth line. The RF classifier is used togheter with the char tokenizer.

Both classifiers in Figure 6.6 displays a MCC score of 0 or close to 0 for all tokenizers except for the char tokenizer in Figure 6.6a up until 140 lines were the performance of the char tokenizer improves. After the red line (all unsuccessful jobs have ended) all combinations are at 0 which is the expected value when using MCC. Only the char tokenizer in Figure 6.6a has an increase in its score the further the job proceeds. The same tokenizer in Figure 6.6b displays the same behavior at first but later on at 370 lines drops back to a score of 0. The other tokenizers were not better with neither of them managing to reach a score higher than 0.5.



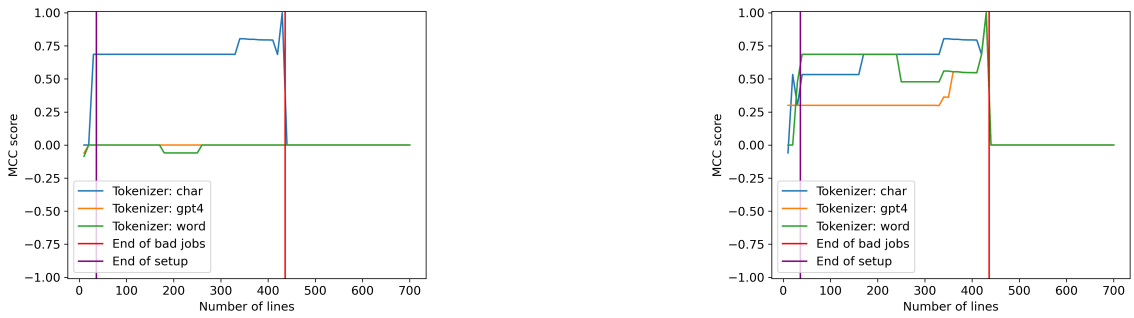
(a) Classifier: RF. **(b)** Classifier: GB.

Figure 6.6: Comparison of different tokenizers on Whole Log model with MCC scores.

Figure B.1 showcases the results of running the simulator on the Whole Log model using three different tokenizers and the classifiers RF in Figure B.1a and GB in Figure B.1b. The results were very scattered when comparing the two classifiers. The char tokenizer had in both cases for RF and GB a sharp rise from 10% accuracy up to between 80% and 90 % but at different times, for RF the rise came at 150 lines whereas the GB saw almost the same rise at 200 lines. Overall the consistency of the accuracy after 150 lines was better for RF. For the GPT4 tokenizer the two classifiers switched places on when one of them was the best, first until 140 lines the GB classifier hold a steady line at 90% meanwhile the RF classifier's accuracy were irregular going from a high of 90%, dropping to 20%, rising to 50% to drop again

after that until it keeps steady at 90%. However, after 280 lines the GB classifier's accuracy falls while the RF classifier remains steady. At around 340 lines both tokenizers exibits the same pattern of first rising and then falling slightly as the number of lines increases. For both classifiers the word tokenizer had an accuracy of 88% until 280 lines. Where the performance took a huge downturn to around 40% for it to rise again sharply after that, this is true for both classifiers as well.

The figures displayed in Figure 6.6 showed the performance of the Whole Log model as it tracked the progression of a real CI job, making predictions over time. Additionally, Figure 6.7 evaluates the MCC score of the Logtime model over time using the simulator. In Figure 6.7a, the RF classifier demonstrated that the char tokenizer outperformed the other two, maintaining a score of 0.7 for most of the CI build. It then reached a perfect score of 1.0 just as the last problematic job concluded. The remaining two tokenizers remained consistently at 0 throughout, with the GPT4 tokenizer experiencing a slight 0.1 drop between lines 180 and 270. In Figure 6.7b, the GB classifier exhibited notably better performance for both word and GPT4 tokenizers compared to RF. The word tokenizer maintained a score of 0.3 until line 320, where it spiked to 0.5 and then to 1.0. Meanwhile, the GPT4 tokenizer performed better than the word tokenizer, scoring 0.6 after the setup had completed, dropping to around 0.45 at line 250, and then following a similar trajectory as the word tokenizer from line 360 onward. However, the char tokenizer initially underperformed compared to RF until line 200, after which its performance mirrored that of the RF classifier.



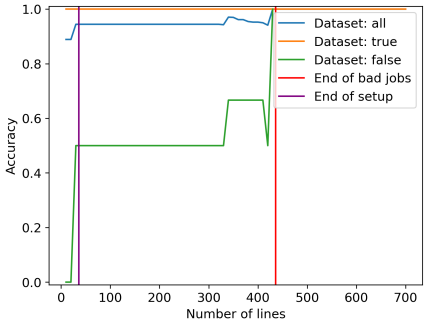
(a) Classifier: RF. **(b)** Classifier: GB.

Figure 6.7: Logtime model's MCC score using several types of tokenizers.

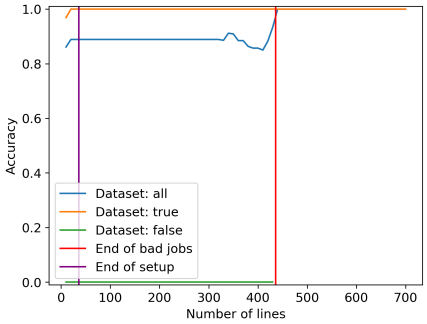
The results depicted in Figure B.1 demonstrated the performance of the Whole Log model as the simulator simulated the progression of a real CI job, making predictions over time. To further evaluate model performance, Figure B.2 presents the results obtained using the Logtime model over time with the simulator. With the Logtime model the accuracy for all tokenizers for both the classifier was much more stable. For the RF classifier in Figure B.2a the char tokenizer performed better than the other two holding about a 10% points accuracy advantage up until only successful jobs were left. Overall the GPT4 tokenizer was slightly better than the word tokenizer between 210 lines and 350 lines, otherwise being identical. In the case of the GB classifier in Figure B.2b the word tokenizer started being best but

after around 280 lines the char tokenizer outperformed it.

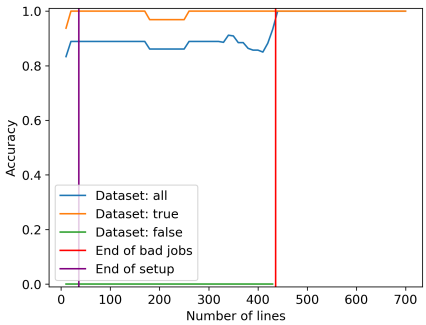
To gain a comprehensive understanding of how various tokenizers classify successful and unsuccessful jobs, six graphs in Figure 6.8 (featuring the Logtime model) and Figure B.3 (depicting the Whole Log model) shows differences in prediction accuracy for all jobs, unsuccessful jobs exclusively, and successful jobs exclusively. Overall the accuracy of predictions for the successful jobs were much better than on unsuccessful jobs in all instances. In two cases with RF and the GPT4 tokenizer in Figure 6.8b respective the word tokenizer in Figure 6.8c had 0% accuracy for the entire duration of the job after the setup process had finished. The char tokenizer however performed better having 100% accuracy on successful jobs throughout as well as 50 to 66% accuracy on unsuccessful jobs. The GB classifier performed better on predicting unsuccessful jobs with the GPT4 tokenizer in Figure 6.8e and with the word tokenizer in Figure 6.8f. In the case of the GPT4 tokenizer this increase came at the cost of lower accuracy when predicting successful jobs dropping from 100% to 95%. The char tokenizer showed lower performance when combined with GB in Figure 6.8d instead of RF as the accuracy on successful jobs was at 95% until the middle of the job, as well as the downward spike in the prediction on unsuccessful jobs at the beginning.



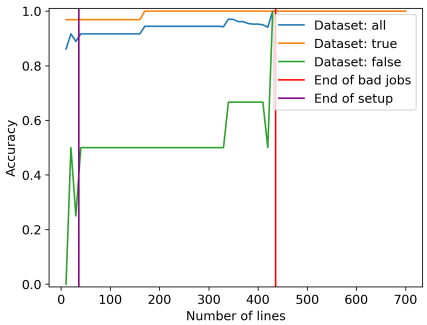
(a) Classifier: RF, Tokenizer: Char.



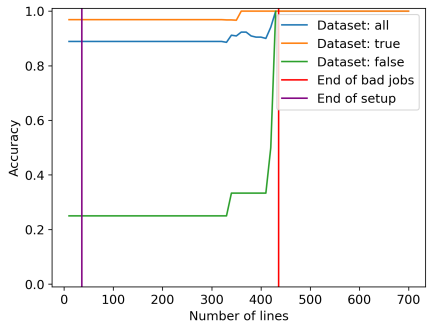
(b) Classifier: RF, Tokenizer: GPT4.



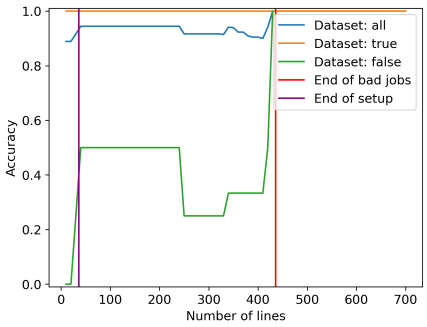
(c) Classifier: RF, Tokenizer: Word.



(d) Classifier: GB, Tokenizer: Char.



(e) Classifier: GB, Tokenizer: GPT4.



(f) Classifier: GB, Tokenizer: Word.

Figure 6.8: Difference between accuracy when predicting unsuccessful and successful jobs running on the Logtime model.

The testing on different datasets in Figures 6.8 showed a distinct pattern at 340 lines tested. After which the accuracy increased and thereafter slowly decreased back until line 420 for it to reach 100% once only successful jobs were left. In order to find out the cause for this Figure 6.9 shows how many incorrect predictions the predictor has made. This indicates a correlation between the quantity of incorrect predictions and the accuracy of the overall prediction. Each instance of incorrect predictions, denoted by the presence of the green, purple, and red lines, corresponds with a rise in accuracy.

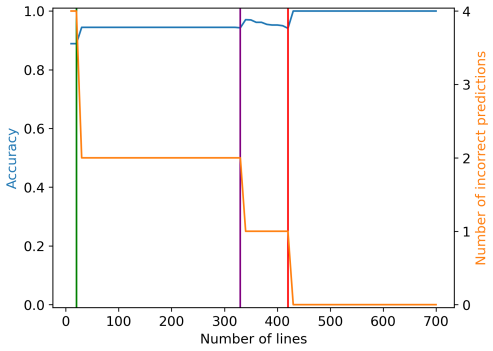


Figure 6.9: Number of incorrect predictions using the RF classifier and char tokenizer where accuracy is represented by the blue line, furthermore the number of incorrect predictions is depicted by the orange line over amount of lines used in prediction.

It is also important to know how much the decrease in number of jobs affect the accuracy and this is depicted in Figure 6.10. No job concludes prior to reaching the 340-line mark, where the first jobs conclude. Following this, multiple jobs conclude amidst the purple and red lines. Beyond the red line, additional jobs conclude, yet the accuracy remains consistently at 100% thereafter. Around the 500-line threshold, a significant number of jobs conclude simultaneously, leaving only four jobs remaining, with the final one concluding at the 710-line mark.

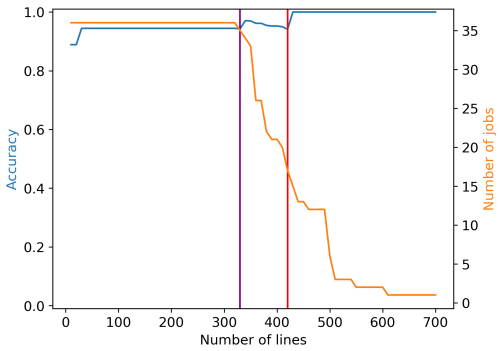
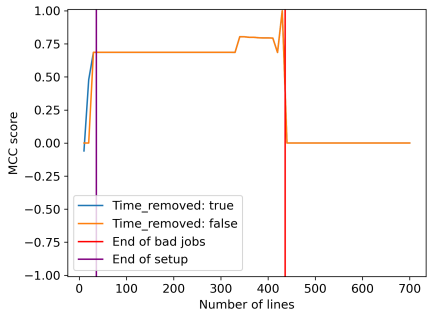
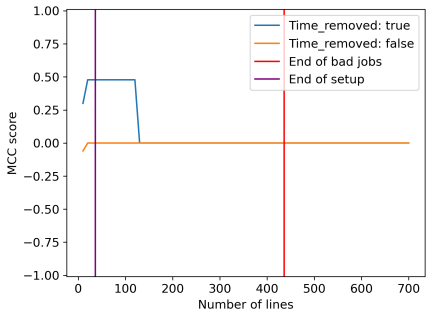


Figure 6.10: Number of jobs that are left after a certain number of lines has been reached. Where accuracy is represented by the blue line, furthermore the number of jobs remaining is depicted by the orange line over amount of lines used in prediction.

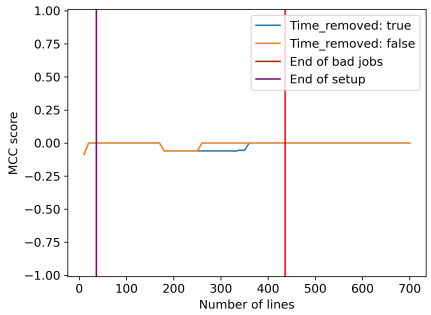
To enhance the generalizability of the logs, an investigation was conducted to assess the impact of timestamps on the results. The differences observed when running the Logtime model are portrayed in Figure 6.11 for MCC score, while those for the Whole Log model are depicted in Figure B.4. The timestamp impacts the MCC score very little, with little impact when running with the RF classifier with examples seen in Figures 6.11a, 6.11b and 6.11c showed no impact as all. In the three cases of running with the GB classifier the timestamp had an bigger impact on the MCC score in Figures 6.11d, 6.11e and 6.11f where the first figure saw a decline in the beginning of the job until line 220. On the other hand the GPT4 tokenizer saw big gains throughout the job were the score increased by a minimum of 0.2 for the entire job until all unsuccessful jobs had passed. The Word tokenizer saw a drop in score of 0.2 between lines 150 and 250 and therafter a lesser drop of 0.1.



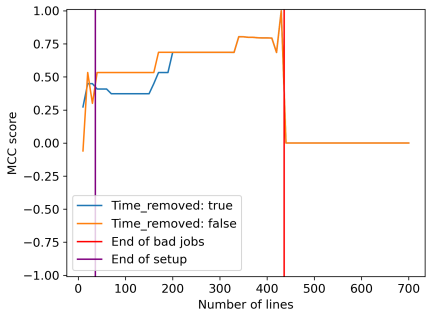
(a) Classifier: RF, Tokenizer: Char.



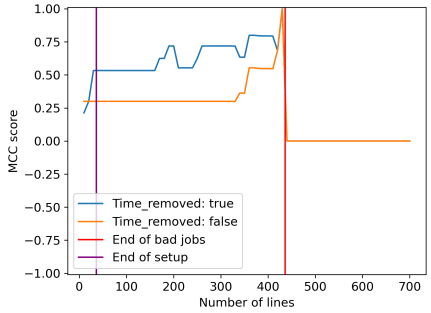
(b) Classifier: RF, Tokenizer: GPT4.



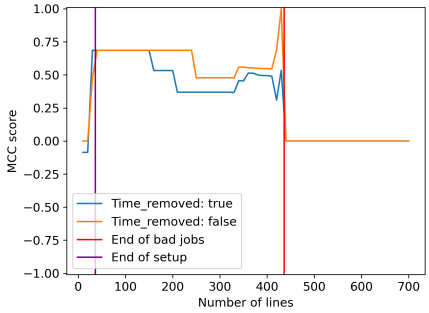
(c) Classifier: RF, Tokenizer: Word.



(d) Classifier: GB, Tokenizer: Char.



(e) Classifier: GB, Tokenizer: GPT4.



(f) Classifier: GB, Tokenizer: Word.

Figure 6.11: Impact of keeping and removing the timestamp from the logs running on the Logtime model.

Furthermore, the differences observed when running the Logtime model are portrayed in Figure B.5 for measuring the accuracy, while those for the Whole Log model are depicted in Figure B.6. The timestamp impacts the accuracy very little where most cases such as Figures 6.11e and 6.11c showed no impact as all and Figure 6.11a showed no impact after the setup phase had ended. In the three cases the timestamp had an impact on accuracy in Figures 6.11b, 6.11d and 6.11f the most effect it had was 5 percentage points.

Table 6.6 presents the RAM usage and training time for each algorithm across all combinations employed in iteration 2, providing insights into the resource requirements of the training process. The Whole Log model's ram usage was the same as the Logtime model for all tokenizers except for the char tokenizer even though the Logtime model had more data points. It instead came at the cost of more time taken to train the Logtime model but this increase varied to a considerable extent depending on the combination of tokenizer and classifier used. The increase was sometime as low as 7% switching from the Whole Log model to the Logtime model with the GPT4 tokenizer on the GB classifier. Wherase the increase when utilizing the RF classifier with the char tokenizer increased by 300% going from the Whole Log model to the Logtime model. Comparing squarely the various tokenizer on only the Logtime model shows that the word tokenizer uses a great amount more RAM than both the other tokenizers staying at 270MB for every combination.

Table 6.6: Performance impact on training with different classifiers, models and tokenizers.

Classifier	Tokenizer	Model	RAM usage (MB)	Time taken (s)
GB	Char	Whole	65	82
GB	GPT4	Whole	83	40
GB	Word	Whole	270	30
GB	Char	Time	83	86
GB	GPT4	Time	83	43
GB	Word	Time	270	37
RF	Char	Time	86	15
RF	GPT4	Time	83	20
RF	Word	Time	270	30
RF	Char	Whole	63	5
RF	GPT4	Whole	83	7
RF	Word	Whole	270	20

6.2.2 Analysis

The focus of the second iteration was on setting up the simulator and running tests that are prevalent to the purpose of the thesis. These results should provide a baseline for what is achievable running these models, tokenizers and classifiers on job 1. It should be noted that the results should improve in forthcoming iterations as there has not been any data balancing or hyperparameter evaluation done yet. This testing will however be done in iteration 3.

Models:
The Whole Log model exhibited inferior accuracy across all tests when comparing the accuracy as can be seen by the differences in accuracy between Figures B.1 and B.2. One potential cause for this descrepancy in accuracy is due to the Whole Log model's inability to capture the contextual information provided by the Logtime model. When faced with shorter logs and using the Whole Log model, the classifier tends to misclassify them as unsuccessful builds, given the typically longer logs associated with successful builds in the dataset used. In Figure B.1a this is clearly seen by the char tokenizer first reaching an acceptable accuracy after 200 lines, and the same is also true for the GPT4 tokenizer. Consequently, the Whole Log model will be omitted in future iterations.

Although the Logtime model demands higher computational resources depicted in Table 6.6 due to its contextual analysis of both the entire log and its segments, it has consistently demonstrated superior accuracy as shown by the differences in accuracy between Figures B.1 and B.2. Considering the computational expense of the Logtime model, an alternative approach involves constructing a model based on the same number of lines as the prediction input, termed the "Lines" model. For instance, if a CI job comprises ten log lines, the corresponding Lines model would be trained solely on the first ten lines of CI logs. This novel approach will be evaluated

against the Logtime model in the forthcoming iteration.

Although the Logtime model almost always outperformed the Whole Log model it is worth noting that the Whole Log model exhibits more expected performance when comparing the accuracy in Figures B.3d and 6.8d. The Logtime model correctly classifies almost 100% of all successful jobs, which is good, however this is expected to be at around 50% accuracy. This is because in the beginning of the job, the successful and unsuccessful job will look the same meaning that it should think some successful jobs may be unsuccessful and vice versa.

The Logtime model was the best because it showed more consistent good performance compared to the Whole Log model. In beginning of the build until the middle point, the Logtime model was always a better performer with the Whole Log only equaling the Logtime model in MCC and accuracy at the end of the build. The Logtime model may take longer to train, however without the MCC or accuracy to boast it becomes pointless to use.

Tokenizers:
The char tokenizer has demonstrated superior performance compared to both word and GPT4 tokenizers when running with the RF model. The char tokenizer performed almost exactly the same in both instances, with a MCC score of 0.7, which is quite good, albeit the performance before line 200 was less stellar as it was close to 0 indicating random predictions. The other two tokenizers performed terribly with RF where predictions were close to random. With the GB classifier the results were more mixed as the char tokenizer were clearly better with the Whole Log model when compared to the word tokenizer and more often than not better than the GPT4 tokenizer. In Figure 6.7b the GPT4 tokenizer was the worst performer while the char and word tokenizers traded blows. Leading to an indication that RF is better with the char tokenizer while GB has not clear winner.

The char tokenizer has also demonstrated superior performance compared to both word and GPT4 tokenizers when looking at the accuracy. Notably, it was the sole tokenizer to achieve an accuracy above 0% when applied to the false dataset with RF, the result can be seen in Figure 6.8a. When applied to the entire dataset, encompassing both successful and unsuccessful job logs, all three tokenizers yielded similar results, with accuracies ranging between 80% and 90% overall for both classes. Consequently, identifying the optimal tokenizer for iteration 3 poses a challenge, as performance can vary depending on the specific characteristics of different jobs. The char tokenizer may be better suited for job 1, given its smaller size relative to the other jobs slated for testing in iteration 3. Additionally, the default hyperparameters used thus far for RF and GB algorithms may favor the char tokenizer over others. Overall the char tokenizer has performed better than the other two it is still too early to draw any conclusion that this is always the case.

Overall, the char tokenizer was the best as it displayed great performance by both having an accuracy of 100% for all successful jobs while simultaneously getting a

good accuracy on unsuccessful jobs. It also received the overall highest MCC score indicating that the char tokenizer is less prone to imbalancing than the rest of the tokenizers.

Classifiers:
Both RF and GB algorithms have exhibited strengths and weaknesses throughout iteration 2. For instance, with certain combinations, such as when using the Logtime model, RF demonstrated superior performance when paired with the character tokenizer in Figure B.2a, whereas GB showed higher accuracy with the word and GPT4 tokenizers, in Figures B.2b. This variability suggests that the choice of classifier may impact the performance of a given tokenizer. Consequently, dropping a classifier could potentially hinder the performance of a particular tokenizer when used in conjunction with other jobs or parameters.

Overall, the best classifier is RF, achieving the highest accuracy and MCC scores while utilizing the fewest resources. It is important to note that these results were obtained in conjunction with the char tokenizer.

Timestamp:
Removing timestamps from job logs during testing revealed that classifiers do not significantly benefit from considering timestamps directly. The impact on the MCC score varied, but overall it is impossible to say that it performed better or worse with the Logtime model in Figure 6.11. While the Whole Log model showed more improvement. As the Logtime model has more data as it splits the log into smaller sections, the timestamp could have a smaller impact as other factors are more important.

The same can be said about accuracy, in some cases, such as Figures B.5b and B.5d, accuracy slightly improved, while in others, like B.5a, B.5c and B.5e it remained unchanged. Interestingly, in one instance (Figure B.5f), accuracy initially dropped before rebounding later. This suggests that classifiers, which only receive tokenized timestamp data, lack the ability to learn from the time intervals between predictions in the simulation.

The MCC score helps in that it can better identify when one class has better accuracy than the other and one such case can be seen on the Whole Log model when comparing the increase in score from B.4e and B.6e where the former sees big improvements until line 370 while the latter only sees a more modest improvements. This indicates that it was one class which got much better accuracy, and this would be the false class as its much more underrepresented.

The assumption made after the initial iteration, therefore, can be dismissed. Several factors may explain why timestamps had a limited effect on the results. For instance, job 1 is executed daily on identical hardware, resulting in nearly consistent completion times for each run, except for unsuccessful jobs. Conversely, jobs with irregular schedules might exhibit different behavior. Consequently, timestamps will

consistently be omitted from training and validation data in subsequent iterations.

Datasets:
The dataset for job 1 exhibited a significant class imbalance, with approximately 89% of the samples representing successful jobs, also shown in Table 6.8. Consequently, with a total sample size of only 174, the hit rate for predicting unsuccessful builds was notably low, while it was substantially high for successful builds better depicted by the graphs in Figure 6.8. This disparity suggests that the predictors trained well on successful data but poorly on unsuccessful job logs. While drawing definitive conclusions from a single job dataset is challenging, this discrepancy may indicate that the models' struggle when faced with datasets where successful jobs vastly outnumber unsuccessful ones.

This shows a clear correlation between the number of falsely predicted jobs and the accuracy. The only thing left unclear is what happens between the purple and red lines in Figure 6.9. One explanation is that it diminishes alongside the reduction in the number of jobs. With the count of unsuccessfully predicted jobs remaining constant as the sample size decreases, the accuracy consequently declines. The number of jobs that are left after a certain amount of lines is depicted in Figure 6.10.

To better validate these findings, it would be beneficial to evaluate the models on datasets that have more samples to validate against. This will be further tested in iteration 3. Additionally, one potential solution to address the imbalance is data resampling, particularly undersampling given the abundance of successful jobs and the scarcity of unsuccessful ones. As oversampling is not possible with the available data, undersampling could help rebalance the dataset and improve model performance. This hypothesis will undergo further testing during iteration 3.

6.3 Third iteration

The focus for iteration 3 was on checking out three shortcomings discussed in the analysis in Chapter 6.2.2 in iteration 2. Namely that evaluation of combinations was done on imbalanced classes, that only default hyperparameters were used and only one job as a data point. In order to overcome these shortcomings, the focus was on balancing the two data classes, using both equal classes but also using different splits in order to see how the MCC score and accuracy is affected. The second focus was on doing hyperparameter tuning as the default hyperparameters for the classifiers are often not the best for every type of classification task. The shortcoming that was evaluated was to test on different jobs as the first job tested on did not have the same amount of data that the jobs used in this iteration had, it was good for gathering lots of data in a brief time. But to get more accurate and generalized data, it is important to also test different types of jobs. All the different combinations evaluated are shown in Table 6.7.

Table 6.7: Combinations evaluated in the third iteration.

Jobs:	Job 1
	Job 2
	Job 3
Classifiers:	RF
	GB
Tokenizers:	Char
	GPT4
	Word
Models:	Logtime
	Lines
Dataset used:	All jobs
	Only unsuccessful jobs
	Only successful jobs
Hyperparameters:	Number of estimators
	Max depth
	Min samples leaf
	Min samples split
Balancing:	All data
	70/30 split
	Equal classes

To grasp the distinctions between the jobs, Table 6.8 provides insights into the quantity of successful and unsuccessful samples for each job, along with the corresponding disk space usage. Job 1 has the lowest amount of samples at only 174 with a size of 16.4 MB. Job 2 was chosen as it has a greater number of samples at 2531 and takes up 450.4 MB. To get an estimation on how large the jobs are the samples are divided by the disk space used. This gives job 1 a mean of 0.094 MB per log and job 2 a mean of 0.178 MB. For job 3 featuring 792 samples and 503.5 MB disk usage, it gives an average of 0.64 MB per log.

Table 6.8: Number of good and bad samples in the different jobs.

Job	Good samples	Bad samples	Size (MB)	Size Equal class (MB)
1	154	20	16.4	2.6
2	2069	462	450.4	222.4
3	691	101	503.5	194.8

In job 2, all samples conform to an identical pattern, save for two outliers. The number of lines when job end are typically between 1300 and 1700, while the time taken can vary considerably. Similar to job 1 as depicted in Figure 6.4, job 2 also exhibits significant variability based on the duration the job waits for a rig to initiate the task. The occurrence of two outliers stems from the inaccurate printing of debug information by rigs reporting their state to the CI machine.

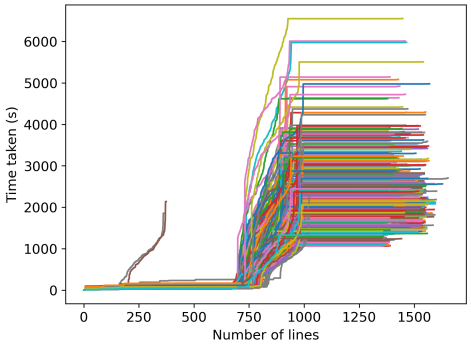


Figure 6.12: Time and line progression for each test sample in job 2.

One reason why the accuracy/MCC scores change the longer the simulator is running is that jobs end at different line counts. For job 2 in Figure 6.13 all jobs stop at almost the exact time, namely after 1400 to 1600 lines. There are some jobs that end at 400 lines, these are outliers from the rest. As a baseline the RF classifier with equal classes and the char tokenizer is used in order to see the contrast between how many jobs are left versus the accuracy is affected.

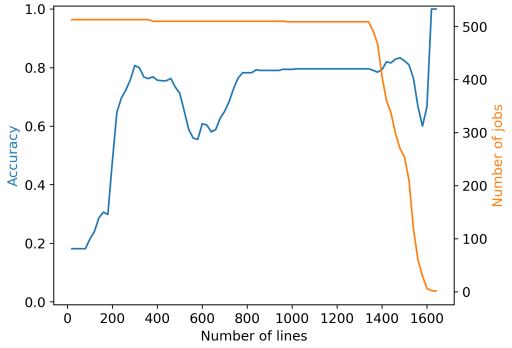


Figure 6.13: Number of jobs that are left after a certain number of lines has been reached in job 2. Accuracy is represented by the blue line as well as the number of jobs remaining depicted by the orange line over amount of lines used in prediction.

Job 3 exhibits significant variability in both the length of its logs and the duration required for completion as shown in Figure 6.14. Similarly, the timing fluctuates considerably based on when a rig connects. Another distinctive aspect of this job, not found in others, is the substantial transformation from its log ending early at 770 lines to the final log ending at 14.510 lines.

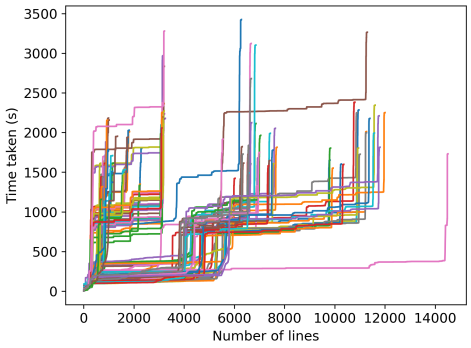


Figure 6.14: Time and line progression for each test sample in job 3.

For job 3 jobs, some jobs end really early at only 1,000 lines, then the number of jobs keep shrinking the longer the job progresses. There are however two points in which no job ended, which are before the purple and red lines in Figure 6.13. During these point it can also be seen that the RF classifier gets affected by the number of jobs still running. Just as in 6.13 the RF classifier is used in conjunction with the char tokenizer and equal classes.

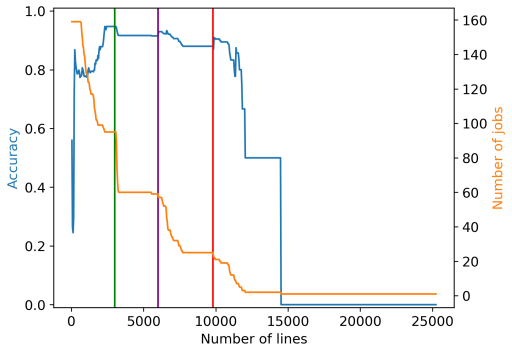
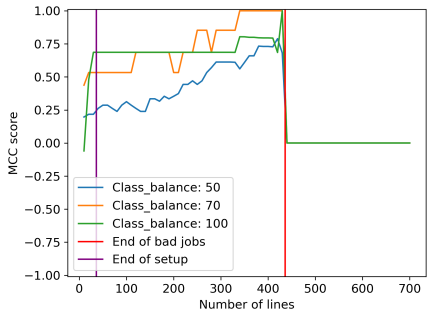


Figure 6.15: Number of jobs that are left after a certain number of lines has been reached in job 3. Where accuracy is represented by the blue line as well as the number of jobs remaining depicted by the orange line over amount of lines used in prediction. The combination used for getting the accuracy metric is RF with the char tokenizer and equal classes.

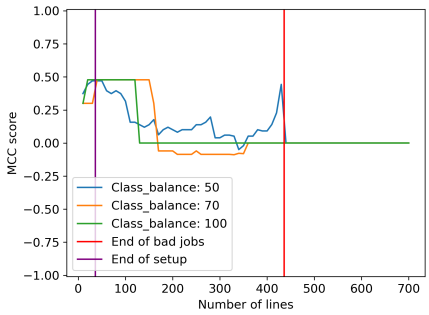
6.3.1 Result

The first part of the results section shows the results from class balancing on the Logtime model, the first Figure, 6.16 displays the MCC score. Overall the score is worse with equal class balancing as is the case in Figures 6.16a, 6.16d, 6.16e and

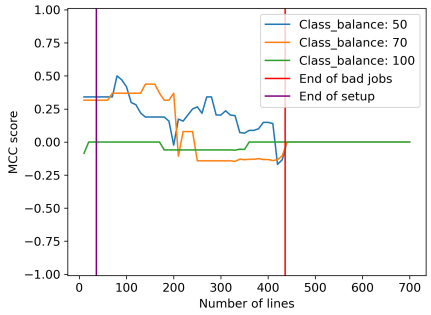
6.16f. What all these have in common is that none of them exceeds the MCC score of when using no balancing on classes at any point in the build process. Figure 6.16c showed gains in the latter half of the CI build, except for the last 20 lines before the unsuccessful jobs end. Gains was also seen in Figure 6.16b, at least for the latter 3/4 of the CI job, although the gains were very minimal of at max 0.10 except for at lines 400 to 420 where the gains was 0.30.



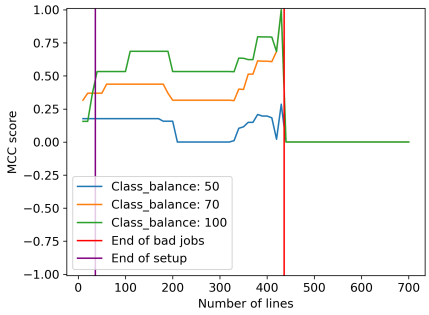
(a) Classifier: RF, Tokenizer: Char.



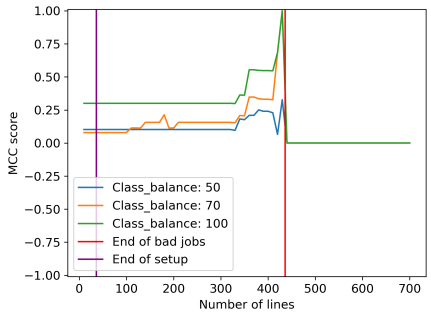
(b) Classifier: RF, Tokenizer: GPT4.



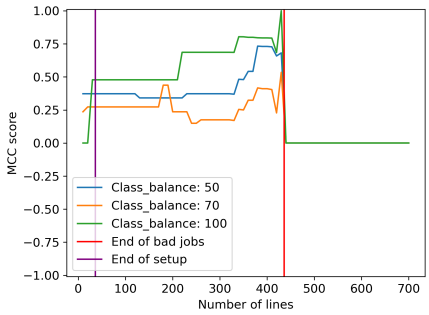
(c) Classifier: RF, Tokenizer: Word.



(d) Classifier: GB, Tokenizer: Char.



(e) Classifier: GB, Tokenizer: GPT4.



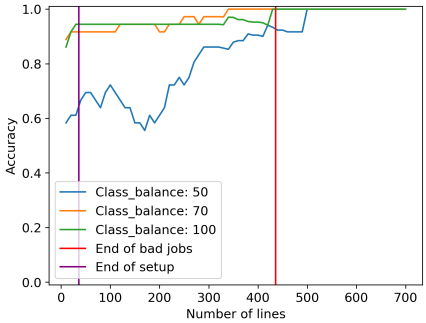
(f) Classifier: GB, Tokenizer: Word.

Figure 6.16: How balancing data on equal classes, 70/30 split in favor of good samples and all samples used affects the MCC score of the different classifiers and tokenizers when running on the Logtime model using MCC as metric.

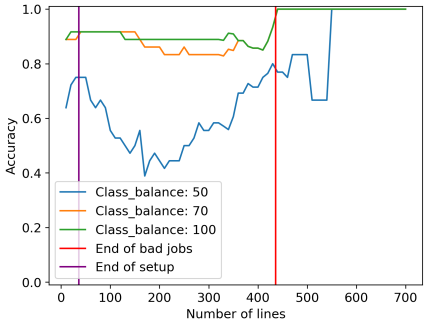
Figure 6.17 illustrates the impact of balancing data through under-sampling of the true class across various tokenizers and classifiers. The under-sampling process involved shuffling the original true training data to generate a new randomized sample. Following the shuffling, the required number of jobs was determined based on the availability of unsuccessful jobs. To show how the prediction of only unsuccessful jobs versus successful jobs differ, this can be seen in Figure 6.18. Both Figures 6.17 and 6.18 features balancing on both classes being equal, as well as 70% which means that 70% of the jobs are good and 30% are bad. The last one is with all samples used, which is the same as in iteration 2.

Figure 6.17 shows that the overall accuracy for all tokenizer and classifier combinations tanked when choosing to balance the classes equally. Having 70% successful and 30% unsuccessful jobs meanwhile showed only a slight decline compared to having all samples available in all cases except for in Figure 6.17f containing the GB classifier and the word tokenizer. Otherwise the 70%, 30% split had only one more outlier in Figure 6.17e after all unsuccessful jobs had ended, it resulted in a decline while all other combinations except for the GB classifier and word tokenizer in Figure 6.17f showed an increase to 100% accuracy.

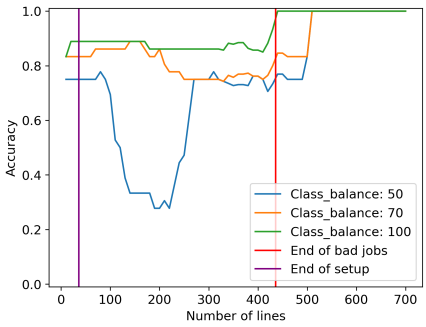
Both char and GPT4 tokenizers with RF in Figures 6.17a and 6.17b respectively climbed steadily as the job got closer to finishing. But the GPT4 tokenizer in combination with RF lost this accuracy gain when all unsuccessful jobs had ended, which ended in a sharp decline to 50% accuracy. Figure 6.17c was the only combination that showed a steep decline in the middle of the CI building process, decreasing sharply from an accuracy of 80% at 110 lines to an accuracy of 20% at 210 lines. Thereafter it made a sharp rise back to 70% at 300 lines. The last two tokenizers char and GPT4 for the GB classifier at equal classes exhibited the same pattern as Figure 6.17f with the 70%/30% split, meaning they hold a steady accuracy until all unsuccessful jobs ended and then the accuracy falls to 0 at the end of the CI build.



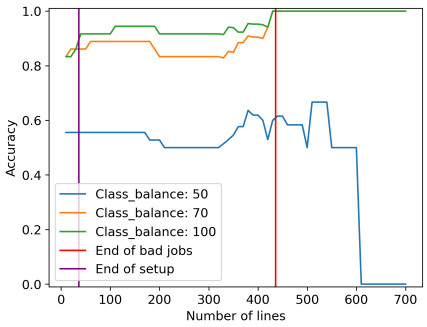
(a) Classifier: RF, Tokenizer: Char.



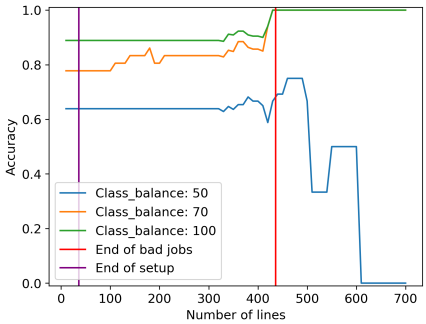
(b) Classifier: RF, Tokenizer: GPT4.



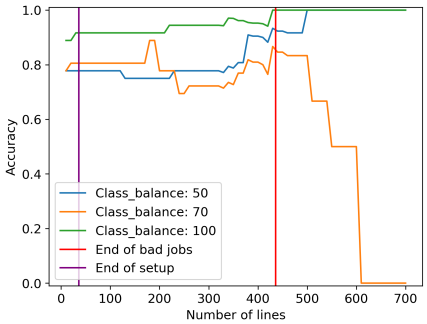
(c) Classifier: RF, Tokenizer: Word.



(d) Classifier: GB, Tokenizer: Char.



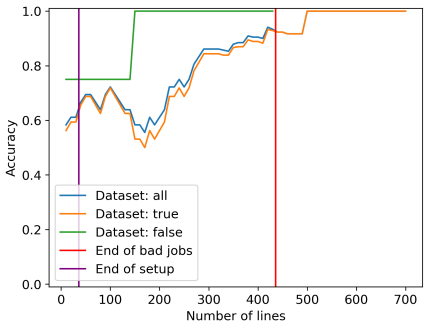
(e) Classifier: GB, Tokenizer: GPT4.



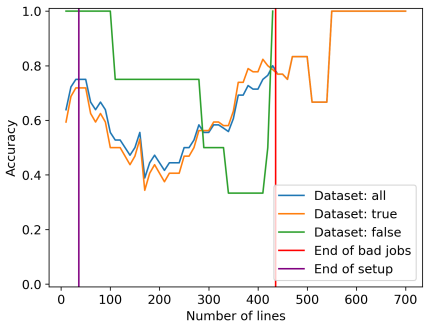
(f) Classifier: GB, Tokenizer: Word.

Figure 6.17: How balancing data on equal classes, 70%/30% split and all samples used affects the accuracy of the different classifiers and tokenizers when running on the Logtime model using accuracy as metric.

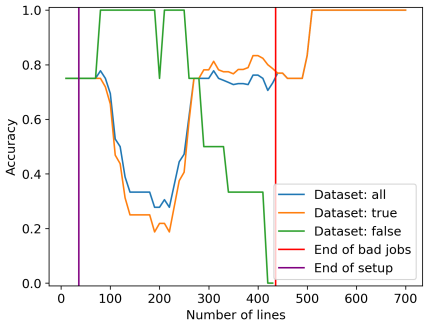
All the results depicted in Figure 6.17 regarding balancing will also be presented in Figure 6.18, offering a more detailed examination of how different datasets are classified, rather than comparing various types of balancing. No balancing, where all samples are used can be observed in Figure 6.8. A preliminary observation reveals that transitioning from equal class distribution to a 70%/30% split does not consistently enhance the accuracy of predicting unsuccessful jobs. Similarly, there is no consistent improvement in accuracy when moving from a 70%/30% split to utilizing all data samples. Conversely, several instances indicate that increasing balancing leads to a decrease in the accuracy of successful jobs, while the accuracy of unsuccessful jobs remains unchanged. For instance, Figures 6.18j, 6.18l and 6.18e exhibit identical predictions for unsuccessful jobs. However, the accuracy of only running on successful jobs demonstrates a decrease in in all three cases.



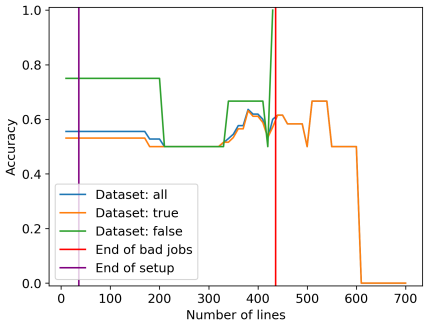
(a) Classifier: RF, Tokenizer: Char, Balancing: 50.



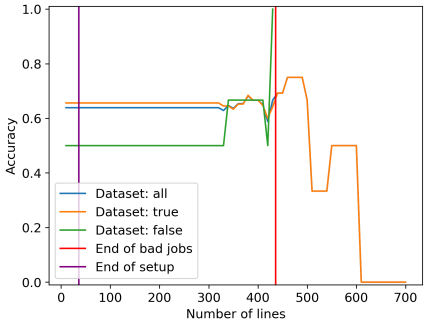
(b) Classifier: RF, Tokenizer: GPT4, Balancing: 50.



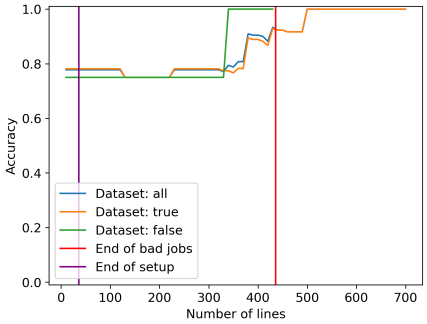
(c) Classifier: RF, Tokenizer: Word, Balancing: 50.



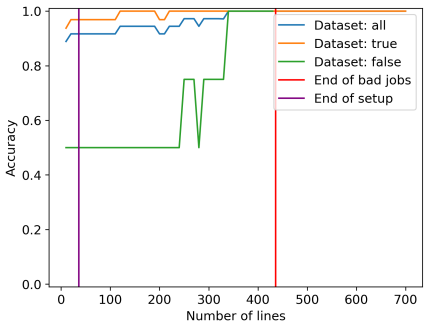
(d) Classifier: GB, Tokenizer: Char, Balancing: 50.



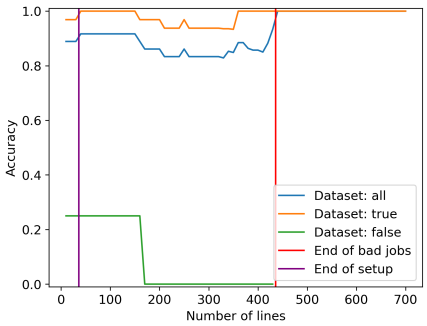
(e) Classifier: GB, Tokenizer: GPT4, Balancing: 50.



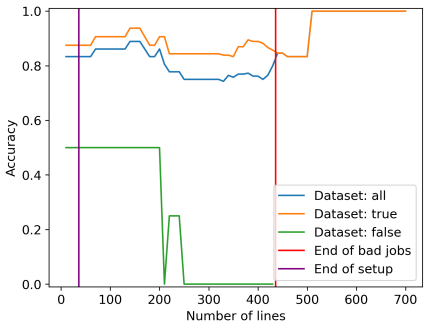
(f) Classifier: GB, Tokenizer: Word, Balancing: 50.



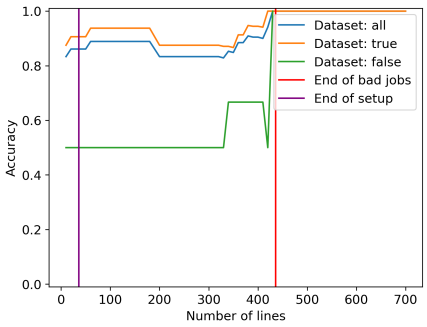
(g) Classifier: RF, Tokenizer: Char, Balancing: 70.



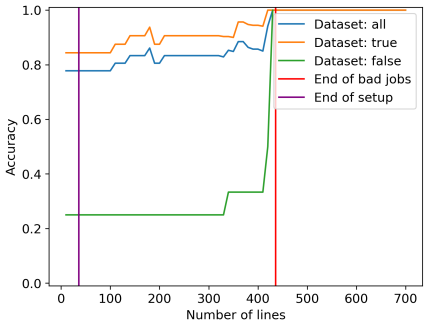
(h) Classifier: RF, Tokenizer: GPT4, Balancing: 70.



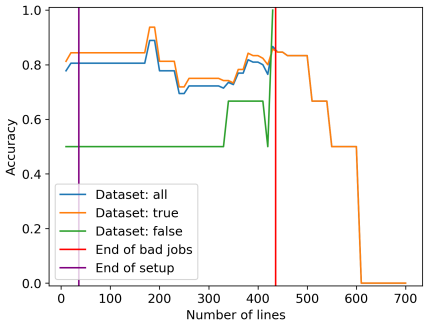
(i) Classifier: RF, Tokenizer: Word, Balancing: 70.



(j) Classifier: GB, Tokenizer: Char, Balancing: 70.



(k) Classifier: GB, Tokenizer: GPT4, Balancing: 70.



(l) Classifier: GB, Tokenizer: Word, Balancing: 70.

Figure 6.18: How balancing data on equal classes and 70%/30% split affects the different classifiers and tokenizers when running on Logtime model.

Both iteration 1, iteration 2 and iteration 3 up until now involved running classifiers with default hyperparameters that were not specifically optimized for the task at hand. To enhance model performance and optimize hyperparameters, GridSearchCV has been utilized on the first job dataset, employing both GB and RF classifiers. The goal was to identify the most effective hyperparameter combinations for improved classifier/model performance. The following parameters was tested:

- n_estimators = [50, 100, 150]
- max_depth = [None, 3, 10]
- min_samples_leaf = [1, 2, 4]
- min_samples_split = [2, 4, 8]

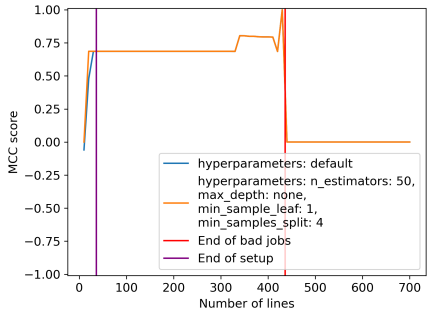
These optimized parameters will then be employed to train and predict the CI build outcome. To further validate the effectiveness of these hyperparameters on subsequent jobs, RandomizedSearchCV has been utilized. Given that the three other jobs are approximately 50 times larger than the first job, running GridSearchCV on all three of them was not feasible due to computational constraints.

The parameters were tested by training all combinations for them to be run in the simulator afterward. The best performing hyperparameters combination, which is shown in Figures 6.19 and 6.20, was determined based on the number of correct predictions during the simulation of a CI job. What this means is that a combina-

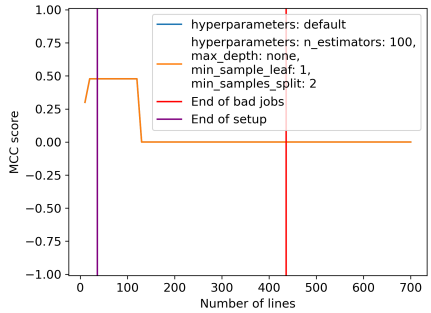
tion of parameters achieving a score of 80% would be labeled the same as achieving a score of 60% in the first half of the job and 100% in the second half of the simulation, as an example.

Hyperparameter tuning increases the MCC score in all cases except for one, that being Figure 6.19b. It shows that testing with different hyperparameters can in some cases significantly increase the MCC score as is the case in Figures 6.19e, 6.19j and 6.19k with the first mentioned figure seeing increases of 0.4 from the end of setup up until line 350, thereafter sowing more modest improvements of about 0.25. The other two in Figures 6.19j and 6.19k showed almost identical improvements where they different slightly as the end of setup up until line 140. Thereafter showing improvements of 0.3 until line 340 where the score increases by almost 1.

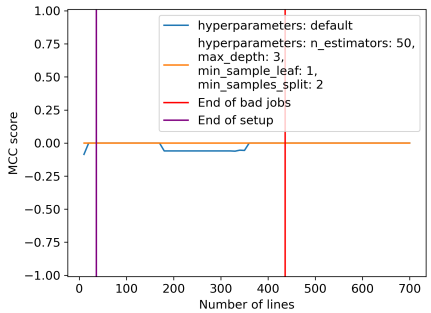
There were also instances where the MCC score did not improve at or or close to 0. This was the case when pairing RF and training using all samples in Figures 6.19a, 6.19b and 6.19c. The biggest improvement was with the word tokenizer with an increase of 0.05 between lines 190 and 380. The char tokenizer did not get any improvement after the setup had finished and the same is also true for the GPT4 tokenizer as well.



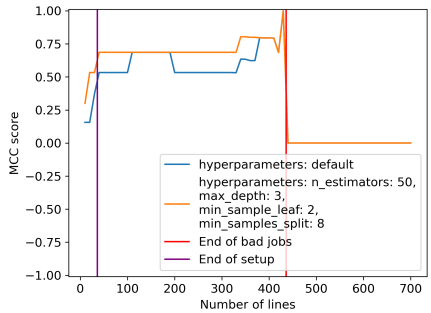
(a) Classifier: RF, Tokenizer: char, Balancing: all.



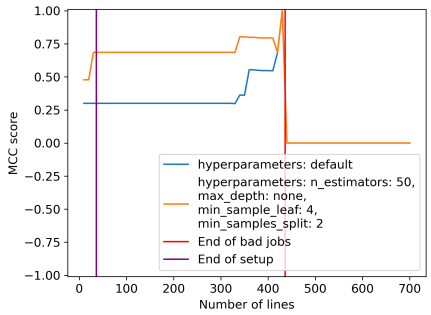
(b) Classifier: RF, Tokenizer: GPT4, Balancing: all.



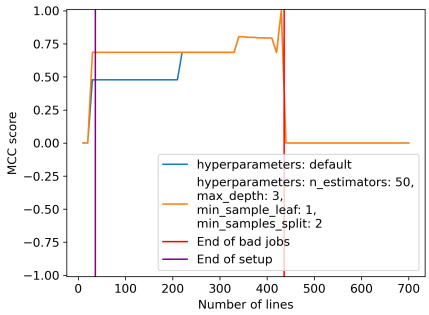
(c) Classifier: RF, Tokenizer: word, Balancing: all.



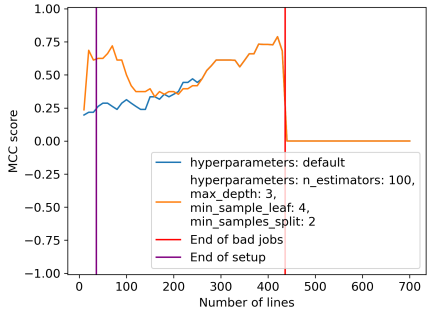
(d) Classifier: GB, Tokenizer: char, Balancing: all.



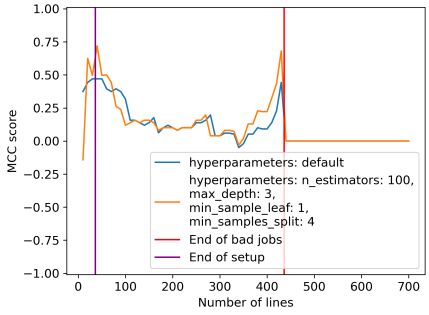
(e) Classifier: GB, Tokenizer: GPT4, Balancing: all.



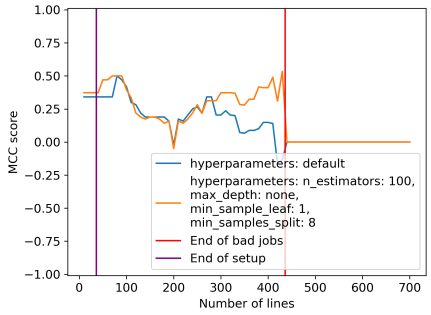
(f) Classifier: GB, Tokenizer: word, Balancing: all.



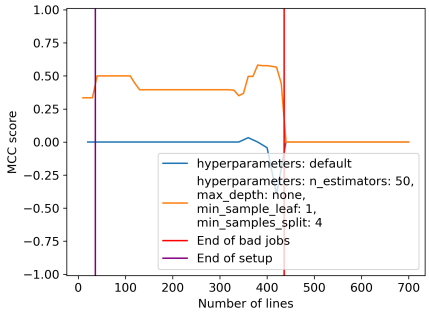
(g) Classifier: RF, Tokenizer: Char, Balancing: equal.



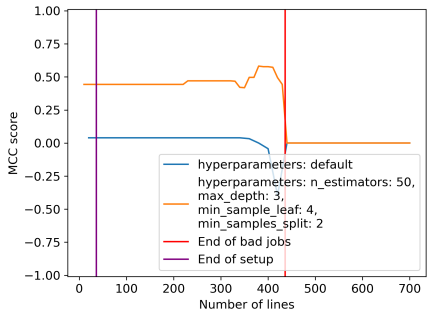
(h) Classifier: RF, Tokenizer: GPT4, Balancing: equal.



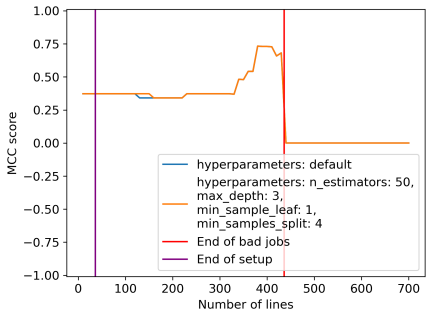
(i) Classifier: RF, Tokenizer: Word, Balancing: equal.



(j) Classifier: GB, Tokenizer: Char, Balancing: equal.



(k) Classifier: GB, Tokenizer: GPT4, Balancing: equal.

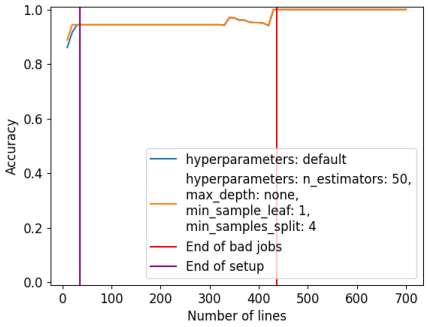


(l) Classifier: GB, Tokenizer: Word, Balancing: equal.

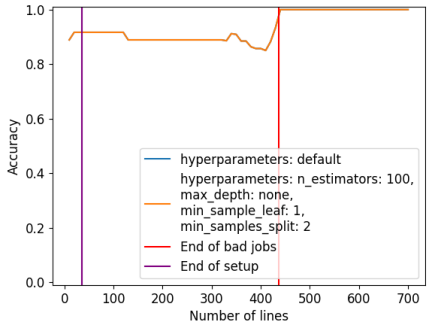
Figure 6.19: GridSearchCV best hyperparameters versus default hyperparameters for all tokenizers and classifiers on the Logtime model on job 1 using MCC as metric.

Accuracy improvement from different hyperparameters depended largely on the classifier used and if balancing on the data had been done. For no data balancing the improvements were minimal with the biggest gain of 5 percentage points seen in Figure 6.20e throughout the CI job up until all unsuccessful jobs had ended. Other combinations saw a small gain of up to 4 percentage points included Figures 6.20c, 6.20d and 6.20f. Meanwhile the combinations of RF with either GPT4 or char tokenizers showed no gain at all after setup was completed.

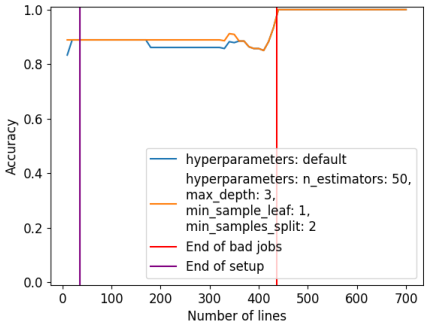
On balancing with equal classes the performance improvements were more noteworthy, and in the case of Figure 6.20j completely changed the result going from 0% all the way to 100% after line 620. Other noteworthy results were Figures 6.20h, 6.20i and 6.20k showing sizable improvements compared to the rest. For 6.20h the biggest improvement is between lines 370 and 500 with a 5 to 10 percentage point increase. A gain can also be seen between lines 370 and 500 for 6.20i as well but the increase is a little more subtle at 4% to 8% point increase. The second largest increase comes from 6.20k showing a 5 to 50 percentage points increase from the beginning of the job up until the 580 line mark has been reached.



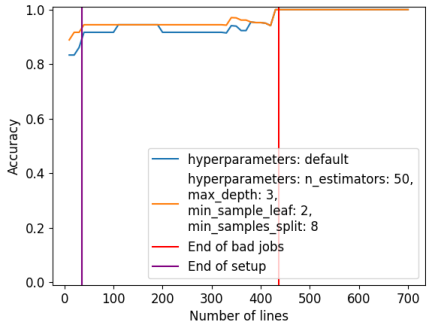
(a) Classifier: RF, Tokenizer: char, Balancing: all.



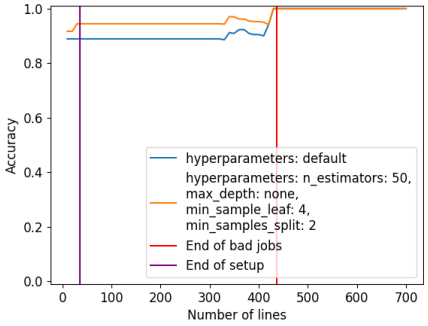
(b) Classifier: RF, Tokenizer: GPT4, Balancing: all.



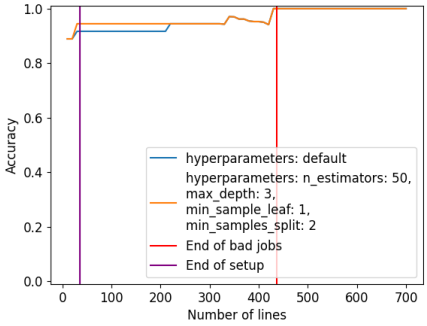
(c) Classifier: RF, Tokenizer: word, Balancing: all.



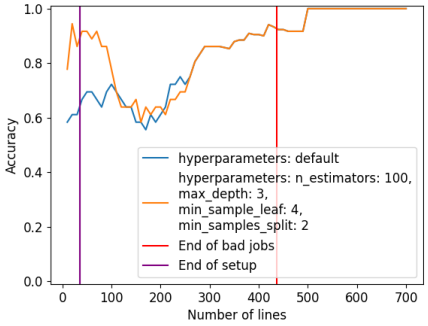
(d) Classifier: GB, Tokenizer: char, Balancing: all.



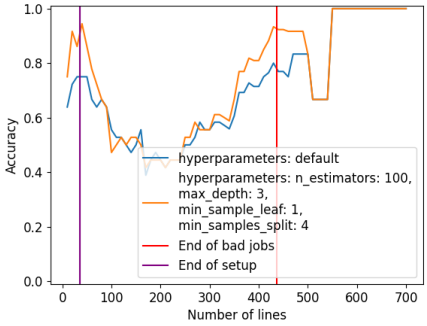
(e) Classifier: GB, Tokenizer: GPT4, Balancing: all.



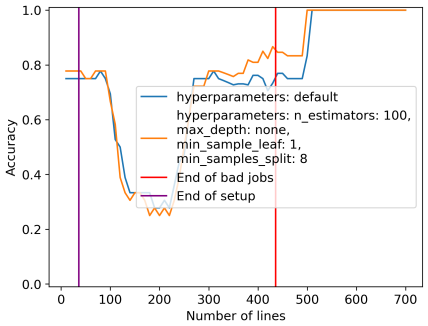
(f) Classifier: GB, Tokenizer: word, Balancing: all.



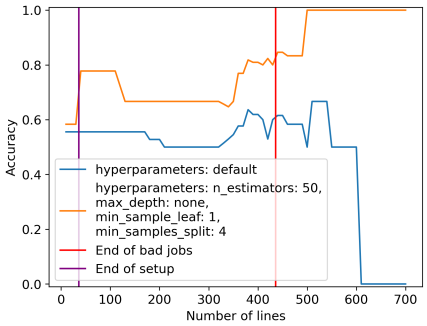
(g) Classifier: RF, Tokenizer: Char, Balancing: equal.



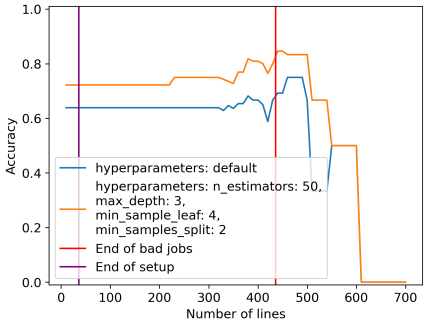
(h) Classifier: RF, Tokenizer: GPT4, Balancing: equal.



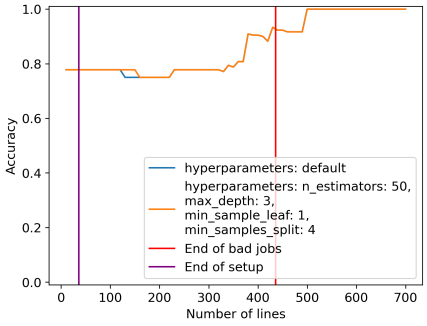
(i) Classifier: RF, Tokenizer: Word, Balancing: equal.



(j) Classifier: GB, Tokenizer: Char, Balancing: equal.



(k) Classifier: GB, Tokenizer: GPT4, Balancing: equal.



(l) Classifier: GB, Tokenizer: Word, Balancing: equal.

Figure 6.20: GridSearchCV best and default hyperparameters for all tokenizers and classifiers on the Logtime model on job 1 using accuracy as metric.

For an understanding on how much the increase actually was in testing with different hyperparameters in Figure 6.20 the following Table 6.9 contains the DTW distance between the two lines compared as well as the compute power needed to run the hyperparameter combination. The results show that in general RF with all samples showed a very low increase as the highest distance was only 0.25 and Figure 6.20b saw no increase at all, the only combination to not see any improvement. GB with all samples saw a higher increase of at the minimum 0.53 and the biggest increase was seen by Figure 6.20e with an increase of 1.68. The biggest gains was all seen with equal classes used. Figure 6.20j saw the biggest distance of 22.32, four times bigger than the second biggest distance of 5.2 in Figure 6.20k. The only combinations to not show close to any improvement at all was Figures 6.20a and 6.20l.

Evaluating hyperparameters involves experimenting with various parameters. Initially, the number of estimators is selected, impacting both results and computational resources. Table 6.9 illustrates the relationship between the best hyperparameters and computational requirements, showcasing optimal combinations identified through GridSearchCV. The next hyperparameter examined is max_depth, with options including "None," 3, and 10. While RF defaults to None, GB defaults to 3, consistent with previous evaluations. When max_depth is set to None, nodes expand until leaves are either pure or contain fewer than the specified minimum samples for splitting. Subsequently, min_samples_leaf is scrutinized against RF and GB's default value of 1, including additional values such as 2 and 4. Finally, min_samples_split is tested, featuring default values of 2 for both RF and GB, alongside alternatives 4 and 8.

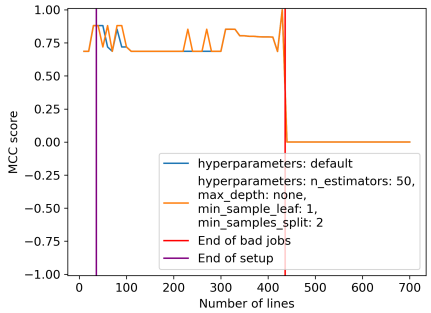
Table 6.9: DTW distance between default and best hyperparameters as well as the performance impact on training with different classifiers, tokenizers and hyperparameters on the Logtime model.

Classifier	Tokenizer	Balancing	N-estimators	Max depth	Min samples leaf	Min samples split	RAM usage (MB)	Time taken (s)	DTW distance
RF	Char	100	50	None	1	4	86	14	0.06
RF	GPT4	100	100	None	1	2	83	18	0.0
RF	Word	100	50	3	1	2	270	30	0.25
GB	Char	100	50	3	2	8	84	50	0.67
GB	GPT4	100	50	None	4	2	83	59	1.68
GB	Word	100	50	3	1	2	270	33	0.53
RF	Char	50	100	3	4	4	63	7	2.44
RF	GPT4	50	100	3	1	4	83	8	2.27
RF	Word	50	100	None	1	8	270	26	1.21
GB	Char	50	50	None	1	4	63	13	22.32
GB	GPT4	50	50	3	4	2	83	13	5.2
GB	Word	50	50	3	1	4	270	20	0.12

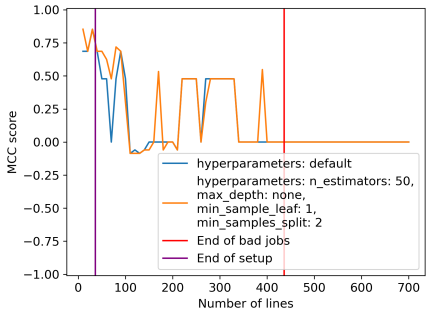
Testing Logtime model for job 1 has concluded this iteration, henceforth all forthcoming figures this iteration showcases the Lines model for job 1. Initially, in Figures 6.21 and 6.22, the MCC score and accuracy for the Lines model was examined using

the same hyperparameters as in iteration 2, alongside the best parameters for the combination used.

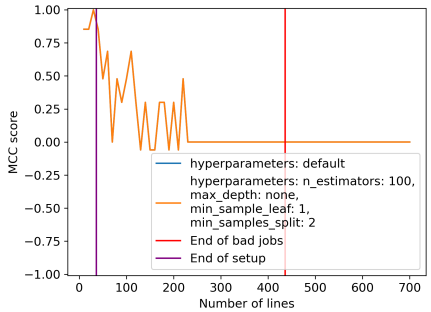
The MCC scores of the Lines model fluctuated significantly, one example is in Figure 6.21c where it started at score of 1 right as the setup completed. then the next prediction moved down to 0.5 to move up to 0.6 then down again to 0, thereafter going up again to 0.4. This pattern then continued of going up and down until line 220 were is stabilized at 0. The same patter of going up and down could be seen across multiple diffent combinations such as 6.21h and 6.21l. Hyperparameter tuning did however help in certain scenarios to lessen this, as was the case in Figure 6.21e.



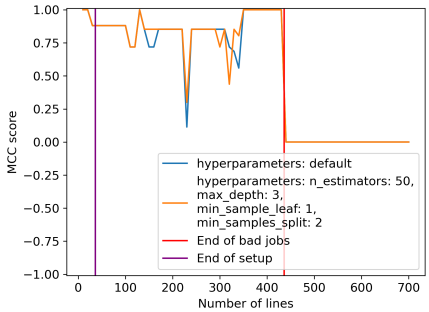
(a) Classifier: RF, Tokenizer: Char, Balancing: all.



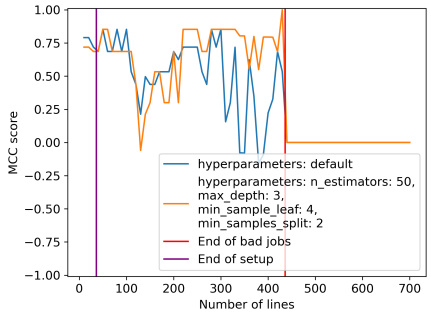
(b) Classifier: RF, Tokenizer: GPT4, Balancing: all.



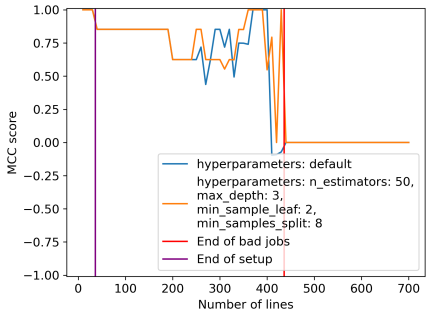
(c) Classifier: RF, Tokenizer: Word, Balancing: all.



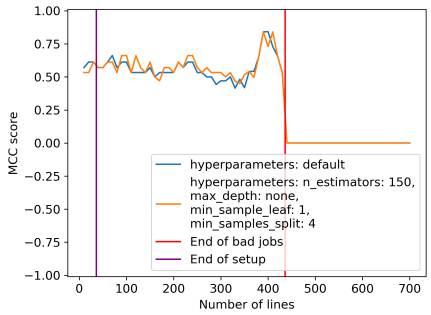
(d) Classifier: GB, Tokenizer: Char, Balancing: all.



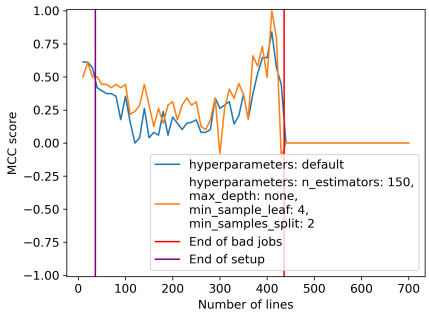
(e) Classifier: GB, Tokenizer: GPT4, Balancing: all.



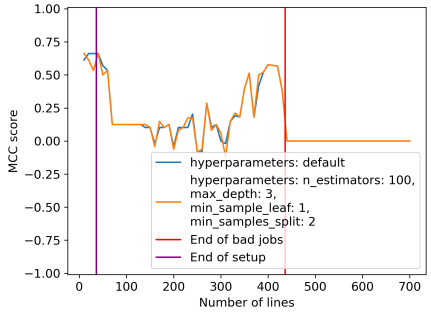
(f) Classifier: GB, Tokenizer: Word, Balancing: all.



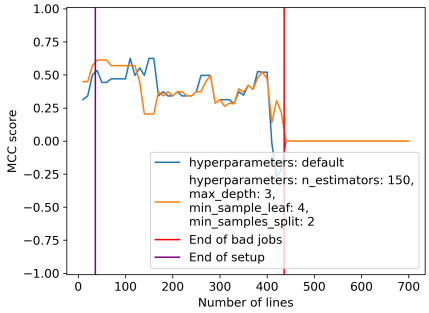
(g) Classifier: RF, Tokenizer: Char, Balancing: equal.



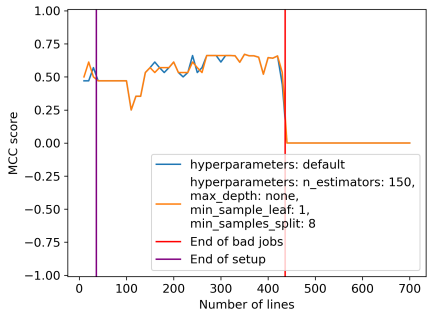
(h) Classifier: RF, Tokenizer: GPT4, Balancing: equal.



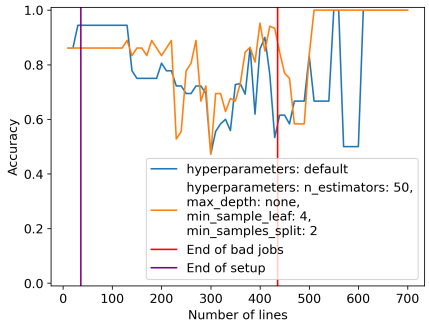
(i) Classifier: RF, Tokenizer: Word, Balancing: equal.



(j) Classifier: GB, Tokenizer: Char, Balancing: equal.



(k) Classifier: GB, Tokenizer: GPT4, Balancing: equal.

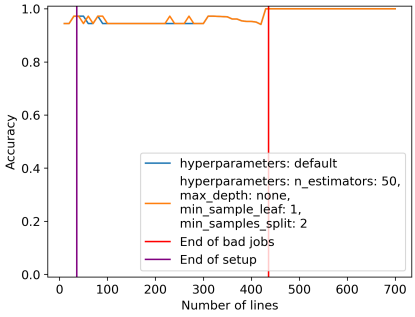


(l) Classifier: GB, Tokenizer: Word, Balancing: equal.

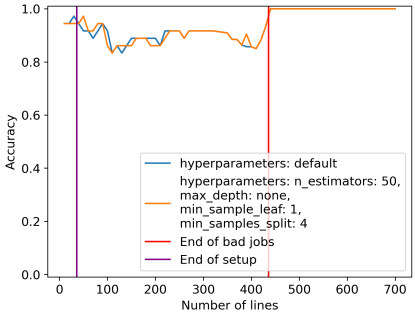
Figure 6.21: GridSearchCV best hyperparameters versus default hyperparameters for all tokenizers and classifiers on the Lines model using MCC as metric.

While the best hyperparameters for RF, executed on all samples in Figures 6.22a, 6.22b, and 6.22c, exhibited marginal improvement, with the maximum improvement observed in 6.22a being 3 percentage points. Figure 6.22c did not demonstrate any accuracy enhancement. In contrast, GB showed more substantial accuracy gains, particularly evident in the elimination of sharp drops in accuracy for 6.22d, achieving better overall accuracy compared to any RF combination. Furthermore, Figure 6.22e mitigated a significant accuracy drop at around 120 lines, rendering the accuracy more stable throughout the build. Once again, default hyperparameters for GB resulted in a drop to 0% accuracy, only to be remedied by the best hyperparameters, ultimately boosting accuracy to 100%.

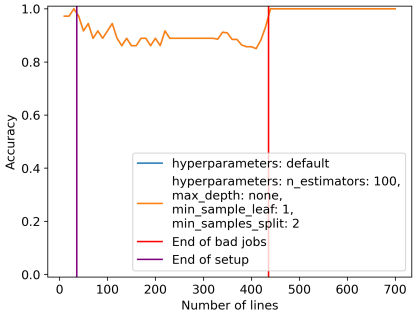
Balancing equal classes significantly influenced the fluctuation of results, causing considerable swings every 10 lines. For instance, Figure 6.22g initially hovered around 80% accuracy but varied between 75% and 90% throughout the job, until all unsuccessful tasks were completed. Subsequently, it swiftly ascended to 100%, only to descend to 50% before returning to 100%. The sole combination that displayed less variability was the GB classifier paired with the GPT4 tokenizer and both classes having the same number of samples in Figure 6.22k. Here, results remained relatively stable, ranging from 75% to 85% for most of the process. At the outset, accuracy started lower, between 60% and 90%, but after 550 lines, it achieved all correct predictions. Conversely, the remaining four combinations in Figures 6.22h, 6.22i, 6.22j and 6.22l exhibited significant accuracy fluctuations to varying degrees. Notably, combinations involving RF maintained higher overall accuracy compared to GB when all classes were balanced.



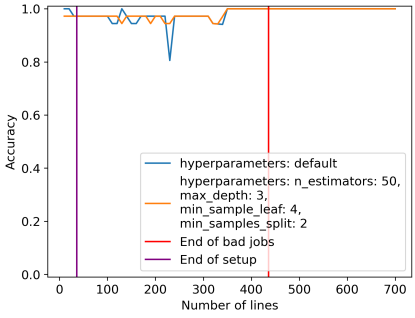
(a) Classifier: RF, Tokenizer: Char, Balancing: all.



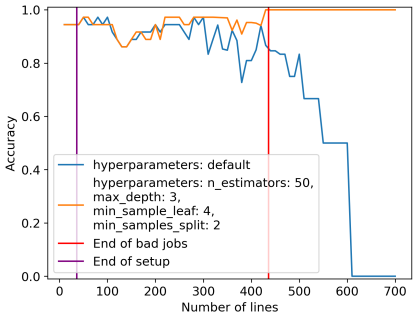
(b) Classifier: RF, Tokenizer: GPT4, Balancing: all.



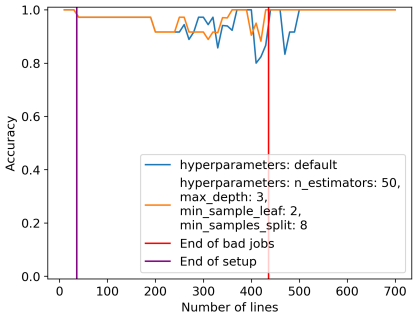
(c) Classifier: RF, Tokenizer: Word, Balancing: all.



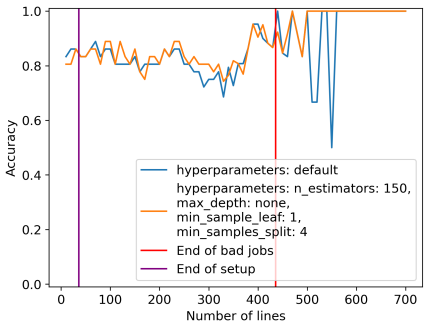
(d) Classifier: GB, Tokenizer: Char, Balancing: all.



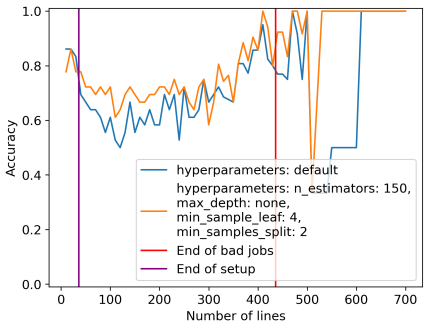
(e) Classifier: GB, Tokenizer: GPT4, Balancing: all.



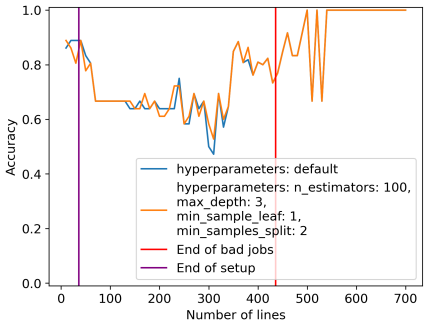
(f) Classifier: GB, Tokenizer: Word, Balancing: all.



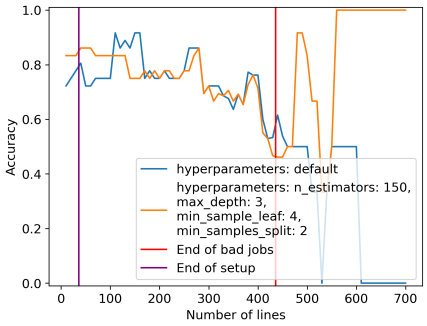
(g) Classifier: RF, Tokenizer: Char, Balancing: equal.



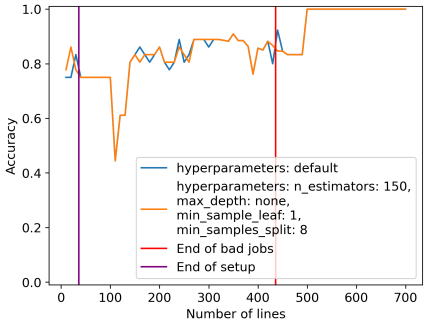
(h) Classifier: RF, Tokenizer: GPT4, Balancing: equal.



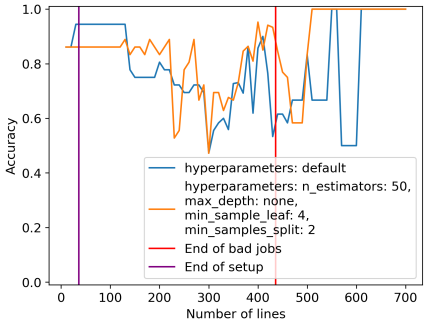
(i) Classifier: RF, Tokenizer: Word, Balancing: equal.



(j) Classifier: GB, Tokenizer: Char, Balancing: equal.



(k) Classifier: GB, Tokenizer: GPT4, Balancing: equal.



(l) Classifier: GB, Tokenizer: Word, Balancing: equal.

Figure 6.22: GridSearchCV best versus default hyperparameters for all tokenizers and classifiers on the Lines model on job 1 using accuracy as metric.

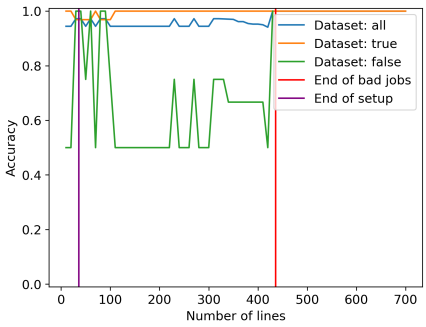
Figure 6.23 illustrates all combinations of tokenizers and classifiers utilized in iteration 3. Half of the graphs, from 6.23a to 6.23f, employ all samples from job 1, while figures from 6.23g to 6.23l utilize balanced classes. In Figure 6.23a, prediction accuracy for successful jobs remains consistently high, always exceeding 90% and never dropping below 100% after 110 lines. However, accuracy for unsuccessful jobs exhibits poorer performance, fluctuating between 50% and 75% initially and then stabilizing at 50% for most of the run. Both Figures 6.23b and 6.23c demonstrate inferior performance, failing to surpass 0% accuracy for unsuccessful jobs for the majority of the build. While occasional spikes in accuracy occur after the first prediction resulting in better performance, subsequent predictions consistently yield

0% accuracy.

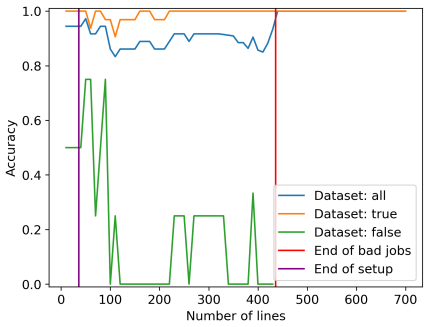
Introducing the GB classifier improves the results for unsuccessful jobs in all three cases. In the first instance, 6.23d, the accuracy of unsuccessful jobs remains at 75% for most of the job run, while the accuracy for successful jobs never falls below 90%. After passing 130 lines, the accuracy remains at 100% for successful jobs with short spikes down to 90% accuracy, whereas unsuccessful jobs sometimes experience dips from 75% accuracy to 50%. For Figure 6.23e the accuracy for the unsuccessful jobs was very sporadic with the accuracy fluctuating between 25% and 75% for the entire run, with two small spikes to 25% as well as one to 100% near the end of the unsuccessful jobs. On the other hand Figure 6.23f had the same 75% accuracy for unsuccessful jobs throughout most of the build between lines 40 and 340. However in the latter two combinations in Figures 6.23e and 6.23f did not feature as good accuracy as there were larger spikes than in 6.23d.

In Figure 6.23g, the results for RF and char tokenizer on equivalent classes revealed that predictions for unsuccessful jobs stayed at 100% during the entire job. However, accuracy for successful jobs fluctuated around 80% during the duration when unsuccessful jobs were not completed. Towards the end, it surged from just below 80% to 100%. Similarly, Figure 6.23h demonstrated a parallel pattern to Figure 6.23g for successful jobs, remaining steady at approximately 75% to 85% accuracy until the completion of unsuccessful jobs. Following that, the combination of RF and GPT4 rose to 90% and remained consistent until around 510 lines, where it reached 100% for the remainder of unfinished jobs after a sharp spike to 35% at the 510 lines mark. Notably, the key distinction between the two lies in their prediction performance on unsuccessful jobs.

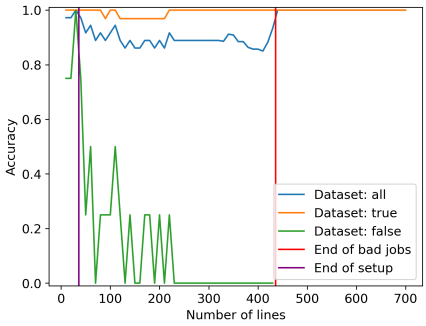
Regarding the word tokenizer in Figure 6.23i, it displayed a similar trend in predicting successful jobs as Figure 6.23h, albeit at a lower percentage, averaging around 70% until making a steep climb to 80% at line 350. Afterwards it rose to 100 % accuracy but only after two more downward spikes to 70% was it able to stabilize at 100%. The green line, however, was more unpredictable, echoing the previous pattern of fluctuation among different predictions made. The sole combination that experienced fewer fluctuations was depicted in Figure 6.23k, initiating successful predictions for successful jobs at 90%. It contained only one large spike down to 40% at 100 lines, this meanwhile unsuccessful jobs stayed at 100% accuracy the entire time.



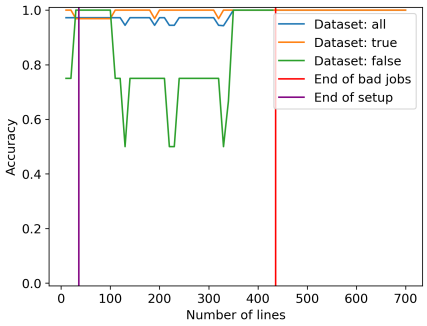
(a) Classifier: RF, Tokenizer: Char, Balancing: all.



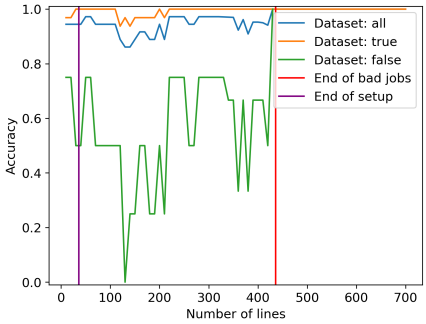
(b) Classifier: RF, Tokenizer: GPT4, Balancing: all.



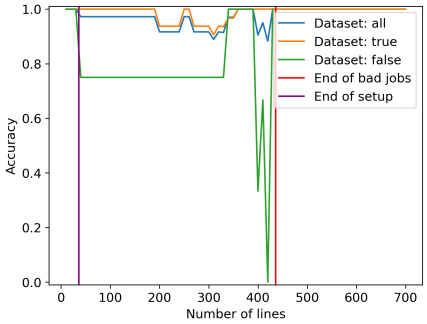
(c) Classifier: RF, Tokenizer: Word, Balancing: all.



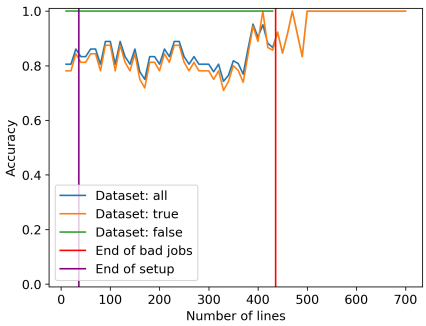
(d) Classifier: GB, Tokenizer: Char, Balancing: all.



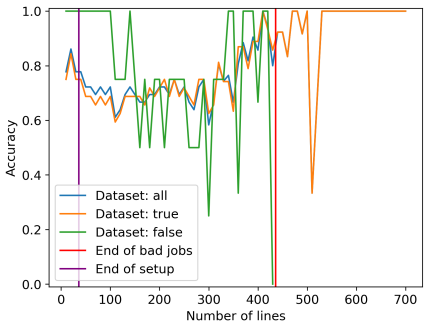
(e) Classifier: GB, Tokenizer: GPT4, Balancing: all.



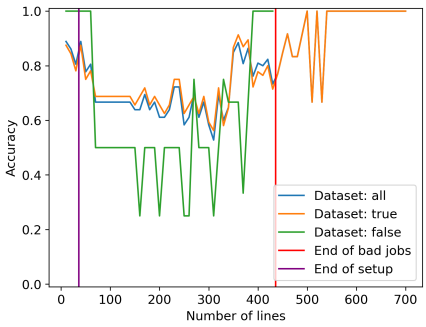
(f) Classifier: GB, Tokenizer: Word, Balancing: all.



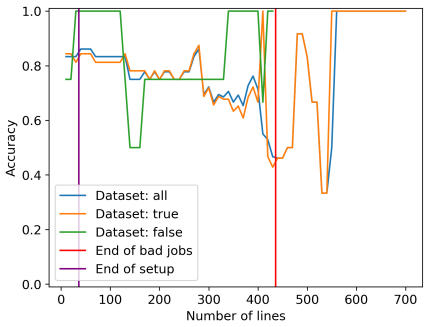
(g) Classifier: RF, Tokenizer: Char, Balancing: equal.



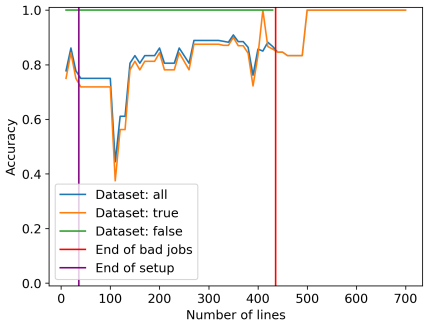
(h) Classifier: RF, Tokenizer: GPT4, Balancing: equal.



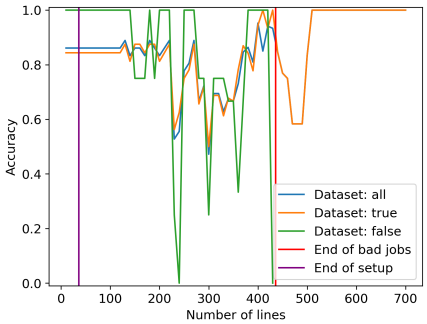
(i) Classifier: RF, Tokenizer: Word, Balancing: equal.



(j) Classifier: GB, Tokenizer: Char, Balancing: equal.



(k) Classifier: GB, Tokenizer: GPT4, Balancing: equal.



(l) Classifier: GB, Tokenizer: Word, Balancing: equal.

Figure 6.23: Best hyperparameters for each combination of classifier and tokenizer on job 1, showing results of predictions on the Lines model for all datasets, successful jobs and unsuccessful jobs using accuracy as metric.

Distance measurement of Figure 6.22 is shown in Table 6.10 that there are bigger gains when using hyperparameter tuning on GB classifier compared to RF. Both Figures 6.22e and 6.22j saw a distance larger than 15. This can however be contrasted to a lot of combinations not getting a distance above 1. Hyperparameters for the Lines model has been evaluated in the same way as with the Logtime model, which results can be found in Table 6.9. Consequently the computational metrics for the Lines model can be found in Table 6.10. The corresponding Figure from where the accuracy distance measured can be found in Figure 6.22.

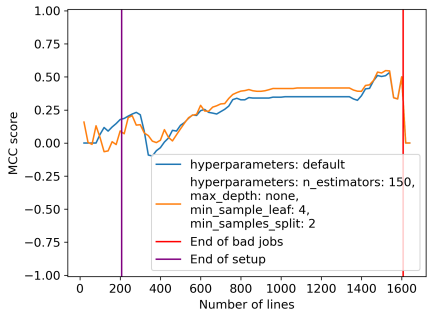
Table 6.10: DTW distance between best hyperparameter and default as well as the performance impact on training with different classifiers, tokenizers and hyperparameters with the Lines model.

Classifier	Tokenizer	Balancing	N-estimators	Max depth	Min samples leaf	Min samples split	RAM usage (MB)	Time taken (s)	DTW distance
RF	Char	All	50	None	1	2	52	210	0.08
RF	GPT4	All	50	None	1	4	154	87	0.18
RF	Word	All	100	None	1	2	270	141	0.0
GB	Char	All	50	3	4	2	210	1641	0.31
GB	GPT4	All	50	3	4	2	154	718	16.75
GB	Word	All	50	3	2	8	270	298	0.96
RF	Char	Equal	150	None	1	4	63	100	1.91
RF	GPT4	Equal	150	None	4	2	83	119	3.47
RF	Word	Equal	100	3	1	2	270	129	0.55
GB	Char	Equal	150	3	4	2	64	1008	16.04
GB	GPT4	Equal	150	None	1	8	83	327	0.40
GB	Word	Equal	50	None	4	2	270	91	5.36

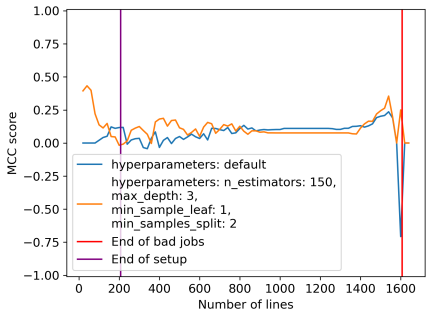
Out of all three jobs selected job 1 has the smallest amount of data when comparing the three jobs. Therefore the rest of this section focuses on job 2 and job 3. All testing has been done with equal classes only meaning that they are not tested using all samples. Furthermore, while GridSearchCV was used on job 1, for job 2 and 3, RandomizedSearchCV has been utilized instead. All parameters are still the same but the number of hyperparameter combinations tested has been reduced in half. Meaning a total of 18 combinations are tested for each tokenizer and classifier combination. One of the combinations evaluated was always the default values meaning that the other 17 combinations are randomized from the parameters used for job 1.

The first testing on job 2 is done using the Logtime model in Figures 6.24 featuring the MCC metrics for all combinations and 6.25 showing the accuracy. Due to the increase in data for the job, the simulator has been changed to have an increase of 20 lines for each prediction instead of 10 that was used for job 1.

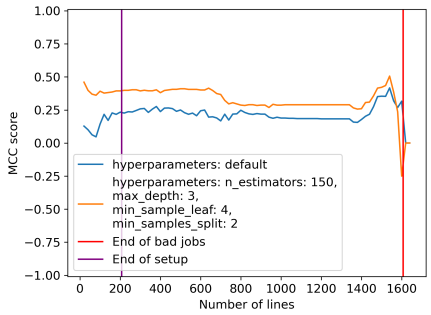
In Figure 6.24, the hyperparameter tuning showed a mostly consistent pattern of increasing the score of about 0.1 to 0.2 throughout all combinations tested. The biggest increase in score was seen in Figure 6.24c although this was from a score of about 0.2 up to 0.35 for the first 750 lines, thereafter staying at 0.3 until 1400 lines. The two combinations that saw the highest score was Figures 6.24d and 6.24f with both of them being around a score of 0.5 during the duration of the build until line 1400 in which both of them rose to 100%.



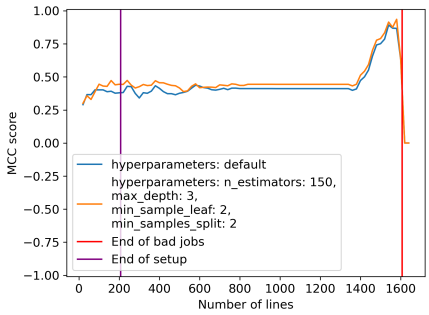
(a) Classifier: RF, Tokenizer: Char.



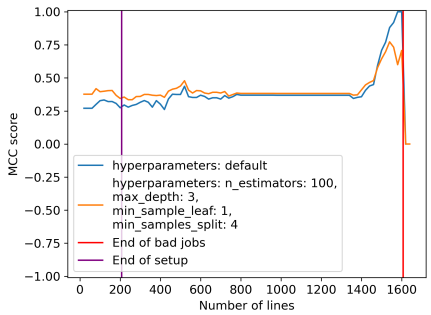
(b) Classifier: RF, Tokenizer: GPT4.



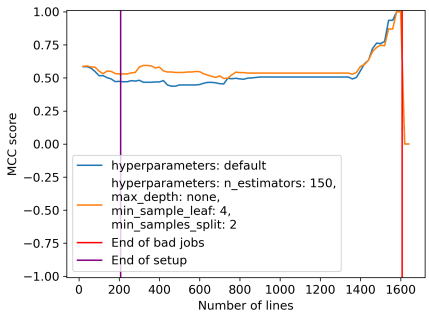
(c) Classifier: RF, Tokenizer: Word.



(d) Classifier: GB, Tokenizer: Char.



(e) Classifier: GB, Tokenizer: GPT4.

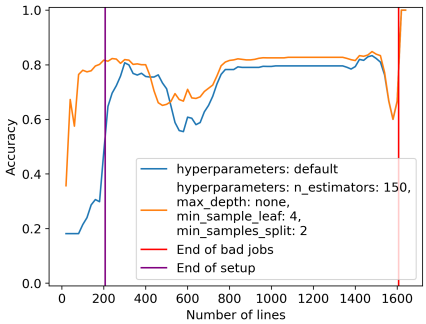


(f) Classifier: GB, Tokenizer: Word.

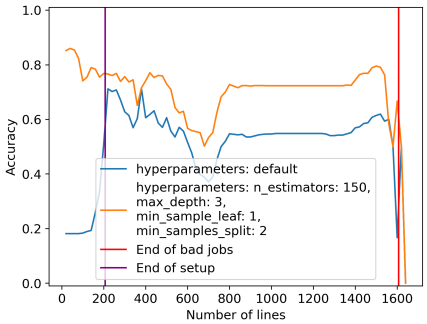
Figure 6.24: Best parameters for each combination of classifier and tokenizer, showing results of predictions on equal classes for the Logtime model running on job 2 using MCC as the metric.

In Figure 6.25f The RF classifier shows more gain with hyperparameter tuning than GB. Although GB already had a higher accuracy compared to RF and thus they both have almost the same accuracy. Figure 6.25c had a big increase in accuracy all through the entire CI job except for the last 100 lines. This meant that at points between 800 lines and 1400 lines the increase was from 65% to 75%. This meant that it had almost the same accuracy as in Figure 6.25e albeit with slightly lower accuracy at the aforementioned point in between line 800 and 1400, before that point, it was however more accurate. Figure 6.25f features the most stable prediction in that it stays at almost the same accuracy, namely 80% for the entire CI build running, only at the end does the accuracy reach 100% when all unsuccessful builds

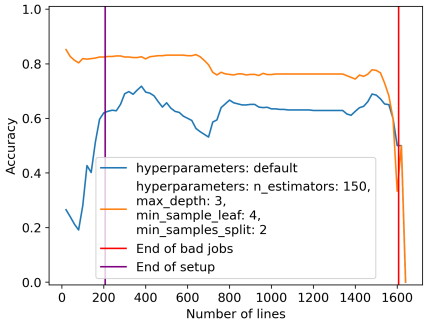
had ended. It is also noteworthy that it was the only build to gain a substantial increase in accuracy at the end. All other combinations had a falloff right before all unsuccessful jobs had ended.



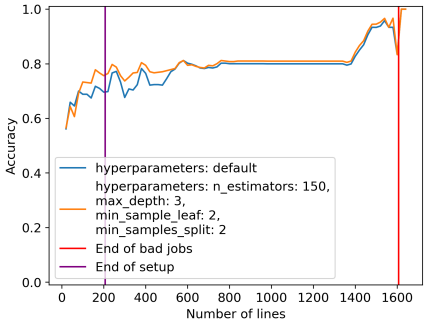
(a) Classifier: RF, Tokenizer: Char.



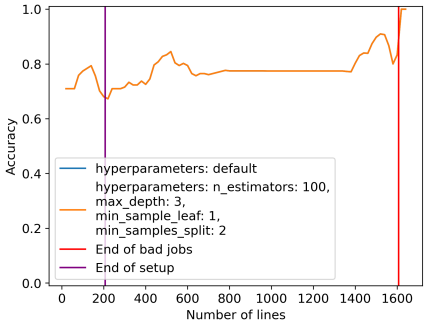
(b) Classifier: RF, Tokenizer: GPT4.



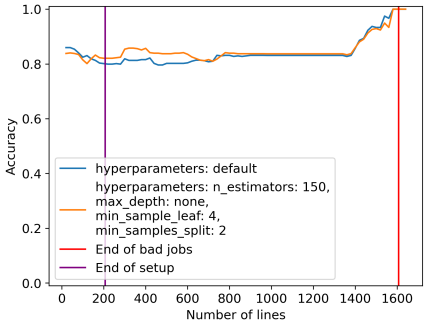
(c) Classifier: RF, Tokenizer: Word.



(d) Classifier: GB, Tokenizer: Char.



(e) Classifier: GB, Tokenizer: GPT4.



(f) Classifier: GB, Tokenizer: Word.

Figure 6.25: Best parameters for each combination of classifier and tokenizer, showing results of predictions on equal classes for the Logtime model running on job 2 using accuracy as the metric.

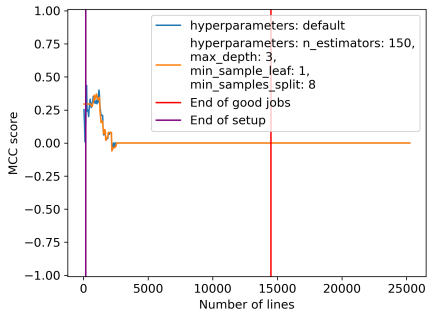
The DTW distance metric for job 2 in Table 6.11 on the Logtime model further shows how the biggest increases are for RF. Distanse for the Hyperparameter tuning on GB showed lesser gains of below 1 and in the case with using the GPT4 tokenizer,

the gain was 0. Performance data shown in Table 6.11 shows a huge discrepancy between training on RF versus GB. The GB classifier took between 10 and 30 time longer to train, these values are for the word and char tokenizers respectively.

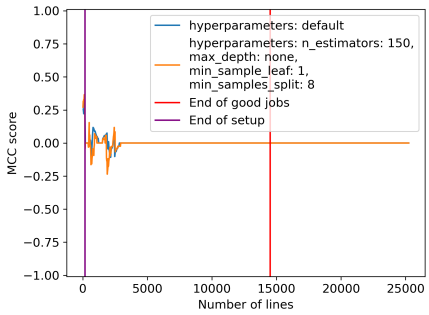
Table 6.11: DTW distance between best and default hyperparameters as well as the performance impact on training with different classifiers, tokenizers and best hyperparameters with the Logtime model on job 2.

Classifier	Tokenizer	Balancing	N-estimators	Max depth	Min samples leaf	Min samples split	RAM usage (MB)	Time taken (s)	DTW distance
RF	Char	Equal	150	3	4	2	9987	125	2.66
RF	GPT4	Equal	150	3	2	2	4644	200	12.86
RF	Word	Equal	100	3	1	2	5024	424	11.71
GB	Char	Equal	150	0	4	2	9987	3633	1.09
GB	GPT4	Equal	150	3	1	8	3417	4318	0.0
GB	Word	Equal	150	0	1	8	5024	4359	0.86

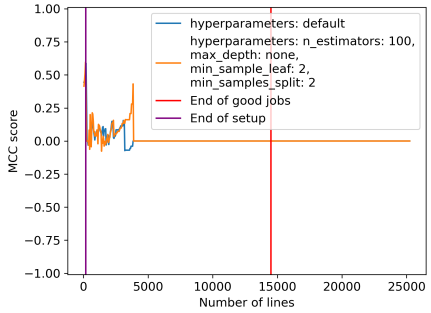
Job 3 is the last job that will be tested, firstly with the Logtime model in Figure 6.26. For job 3 it is important to take note of the fact that the red line do not indicate the end of unsuccessful jobs but instead the end of successful jobs. This is because there are unsuccessful jobs that are have more log lines than every successful build. The result showed that RF managed to perform better than a score of 0 for the first 3,000 lines in all three cases. However, afterward it stayed at 0 for the rest of the build. The GB classifier in Figure 6.26d displayed that it managed to stay above 0 up until shortly after 5,000 lines had been reached, in which point the score was 0 and remained there for the rest of the build. This was however not the case with default parameters in which the score was below 0 from 500 lines until around 13,000 lines. Figure 6.26e illustrated excellent performance from line 12,000 to the end of good jobs with a perfect score of 1. At the beginning of the job up until line 12,000 it instead scored at around 0.2.



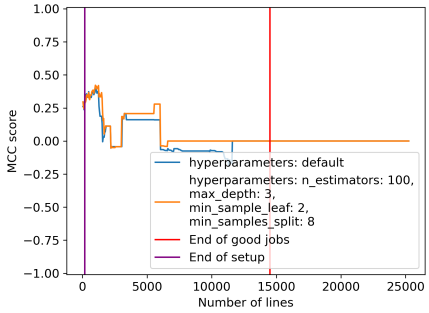
(a) Classifier: RF, Tokenizer: Char.



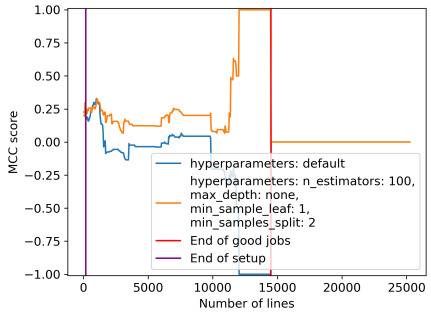
(b) Classifier: RF, Tokenizer: GPT4.



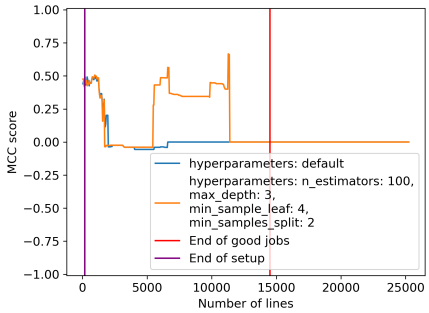
(c) Classifier: RF, Tokenizer: Word.



(d) Classifier: GB, Tokenizer: Char.



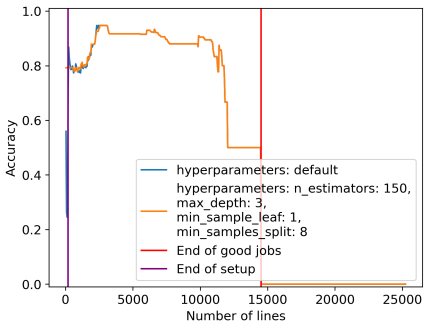
(e) Classifier: GB, Tokenizer: GPT4.



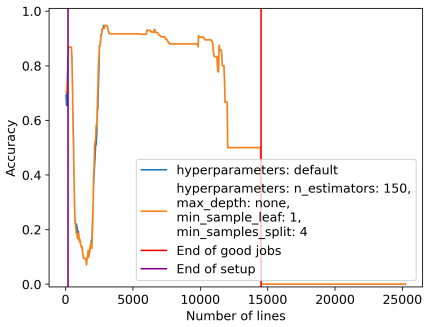
(f) Classifier: GB, Tokenizer: Word.

Figure 6.26: Best parameters for each combination of classifier and tokenizer, showing results of predictions on equal classes for the Logtime model running on job 3 using MCC as metric.

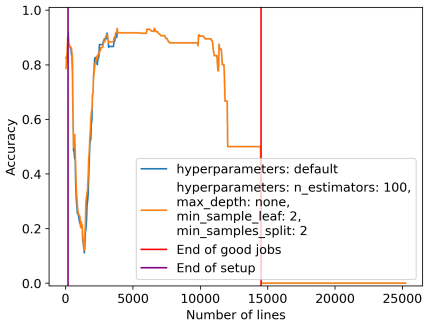
At a first glance the different graphs in Figure 6.27 shows different patterns. The first is that RF together with either GPT4 in Figure 6.30b and word tokenizer in Figure 6.30c shows almost the same result. The biggest difference being 5% at one single point at around 3,000 lines. All figures except for Figure 6.27e saw its accuracy drop from 80% to exactly 50% at 12,000 lines. Then when all unsuccessful jobs have ended at 15,000 lines they all fall to 0%. The only exception to this was Figure 6.27e showing an an increase from 60% to 100% instead of a drop to 50% like the others. This was however only when factoring in using the best hyperparameters, otherwise it went straight to 0%.



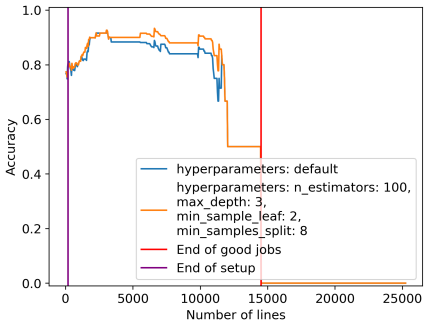
(a) Classifier: RF, Tokenizer: Char.



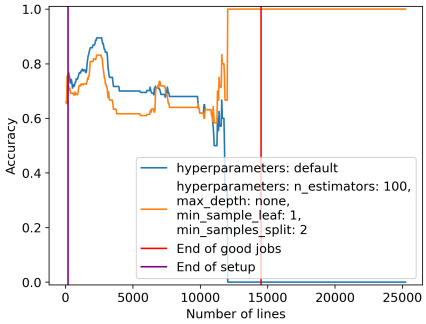
(b) Classifier: RF, Tokenizer: GPT4.



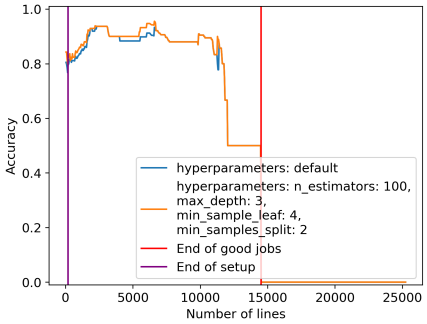
(c) Classifier: RF, Tokenizer: Word.



(d) Classifier: GB, Tokenizer: Char.



(e) Classifier: GB, Tokenizer: GPT4.



(f) Classifier: GB, Tokenizer: Word.

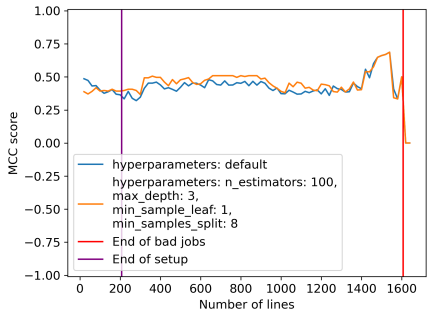
Figure 6.27: Best parameters for each combination of classifier and tokenizer, showing results of predictions on equal classes for the Logtime model running on job 3 using accuracy as metric.

Table 6.12 illustrates that the distance of the hyperparameters is larger than it was for job 2 in Table 6.11. It also highlights the massive increase from 0% to 100% in Figure 6.27e. Furthermore for the Logtime model on job 3 is the performance impact which is also displayed in Table 6.12. This shows a giant discrepancy between how long it takes between the RF and GB classifier for their best hyperparameters. When training using the char tokenizer RF was 56 times faster compared to GB. The same trend is also seen with the GPT4 and Word tokenizers where the RF classifier is 10 respective 2.9 times faster.

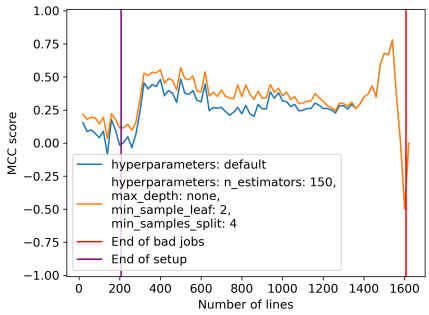
Table 6.12: Performance impact on training with different classifiers, tokenizers and hyperparameters with the Logtime model on job 3.

Classifier	Tokenizer	Balancing	N-estimators	Max depth	Min samples leaf	Min samples split	RAM usage (MB)	Time taken (s)	DTW distance
RF	Char	50	150	3	1	8	9434	193	2.27
RF	GPT4	50	150	None	1	4	4352	488	1.32
RF	Word	50	100	None	2	2	5159	748	1.13
GB	Char	50	100	3	2	8	9434	10711	3.16
GB	GPT4	50	100	None	1	2	4351	4877	344.28
GB	Word	50	100	3	4	2	5159	2127	1.59

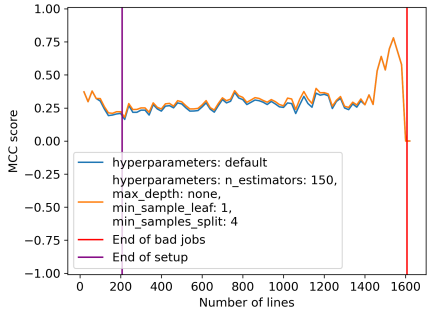
Running the Lines model on job 2 depicts a small increase in the MCC score in Figure 6.29. The two biggest increases are with RF in Figures 6.28a and 6.28b. The other four Figures depicts less increases in the MCC score and in Figure 6.28f there is a slight decrease in the MCC score, which means that the chosen hyperparameter increases the imbalance between correct predictions on successful and unsuccessful jobs. Figure 6.28d did show a better MCC score but only because of what happens at line 1200 to 1400, otherwise the decrease in the score at line 200 to 1,000 would have meant this combination also got a slight drop in MCC score.



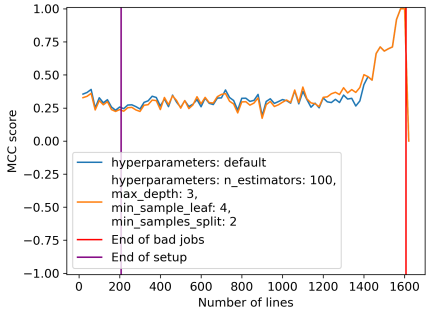
(a) Classifier: RF, Tokenizer: Char.



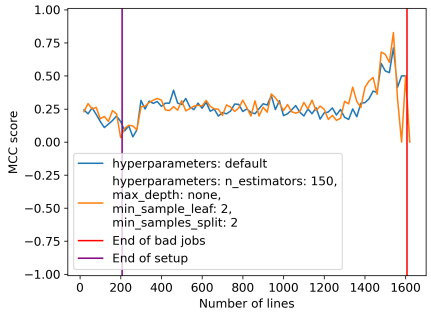
(b) Classifier: RF, Tokenizer: GPT4.



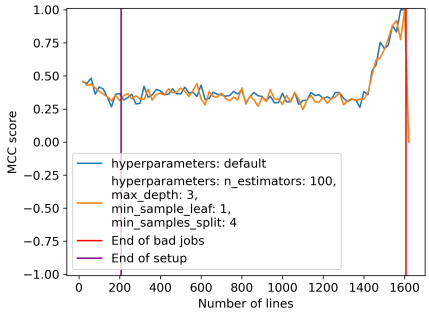
(c) Classifier: RF, Tokenizer: Word.



(d) Classifier: GB, Tokenizer: Char.



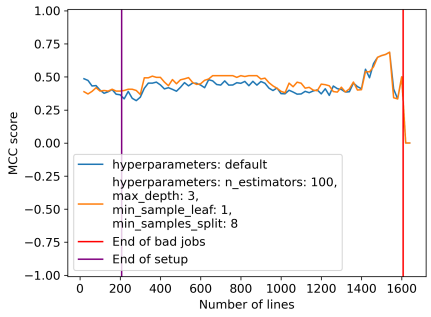
(e) Classifier: GB, Tokenizer: GPT4.



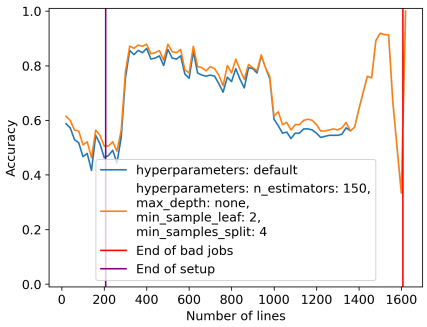
(f) Classifier: GB, Tokenizer: Word.

Figure 6.28: Best hyperparameters for each combination of classifier and tokenizer, showing results of predictions on equal classes for the Lines model running on job 2 using the MCC metric.

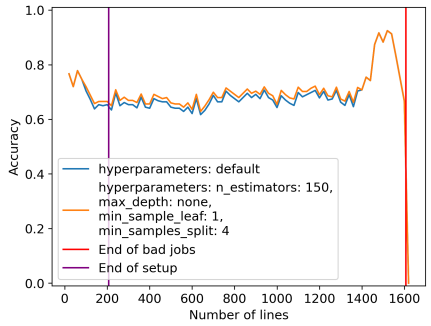
Compared to the MCC score the accuracy shown for the Lines model on job 2 in Figure 6.29 shows bigger increases. The biggest increase is with the GB classifier and the char tokenizer in Figure 6.29d were at line 1200 to 1400 with a climb of 10% points. The accuracy boost before was also an increase, but with about 5% points instead. Although the accuracy gain was larger than the MCC score, it was still only by about 5% points in Figures 6.29a, 6.29b and 6.29c. Meanwhile Figure 6.29f had sporadic gains and also drops below the accuracy of the default hyperparameters. This meant that the total increase in accuracy when accounting for all prediction points was just above 0%.



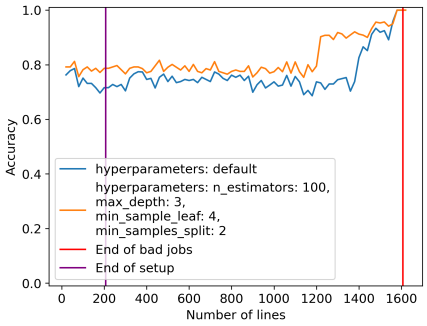
(a) Classifier: RF, Tokenizer: Char.



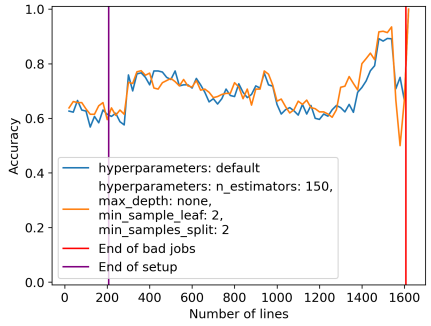
(b) Classifier: RF, Tokenizer: GPT4.



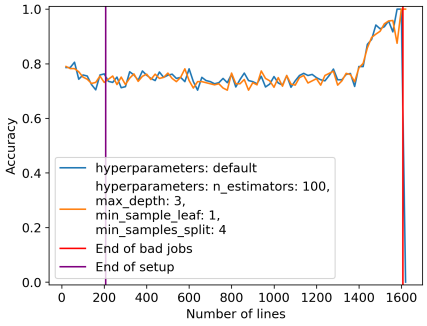
(c) Classifier: RF, Tokenizer: Word.



(d) Classifier: GB, Tokenizer: Char.



(e) Classifier: GB, Tokenizer: GPT4.



(f) Classifier: GB, Tokenizer: Word.

Figure 6.29: Best parameters for each combination of classifier and tokenizer, showing results of predictions on equal classes for the Lines model running on job 2 using the accuracy metric.

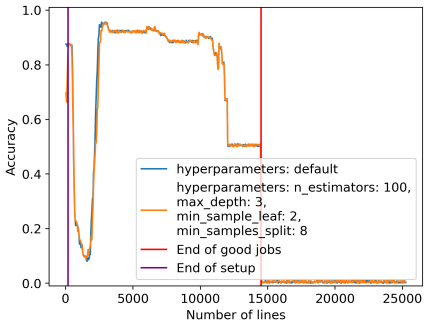
The DTW distance metric reveled that all combinations evaluated in Figure 6.29 does not gain much from Hyperparameter tuning. The results in Table 6.13 shows the biggest gain is 1.49 but this is only for one combination, meanwhile all others are below a distance of 1. Table 6.13 then present the computational power needed to train the Lines model. The previous findings on the GB classifier always taking longer than RF is further solidified. This time GB is only 14 times slower than RF at max.

Table 6.13: DTW distance between the best hyperparameters and the default hyperparameters as well as the performance impact on training with different classifiers, tokenizers and hyperparameters with the Lines model on job 2.

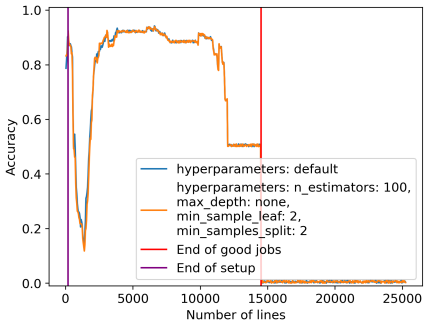
Classifier	Tokenizer	Balancing	N-estimators	Max depth	Min samples leaf	Min samples split	RAM usage (MB)	Time taken (s)	DTW distance
RF	Char	Equal	100	3	1	8	7914	1005	1.49
RF	GPT4	Equal	100	3	1	4	4292	598	0.79
RF	Word	Equal	100	3	1	4	4098	1898	0.64
GB	Char	Equal	100	3	1	2	9234	14127	0.98
GB	GPT4	Equal	150	None	2	2	3485	6585	0.0
GB	Word	Equal	100	3	1	4	6502	4255	0.87

The Lines model in Figure 6.30 illustrates that there are small variations between the predictions as the lines is jittery and not smooth. Apart from that the result are almost exactly the same as in Figure 6.27.

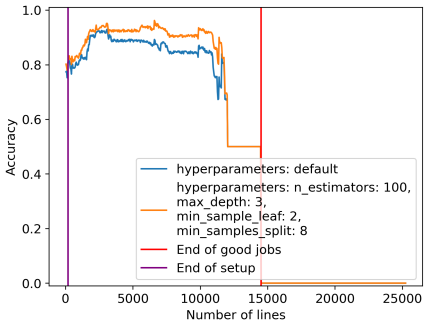
(a) Classifier: RF, Tokenizer: Char



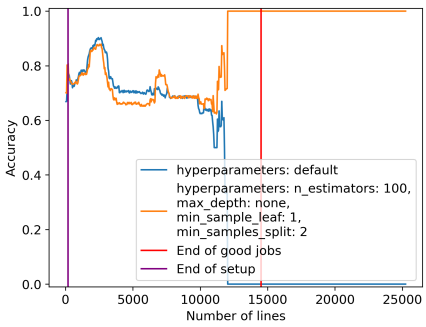
(b) Classifier: RF, Tokenizer: GPT4



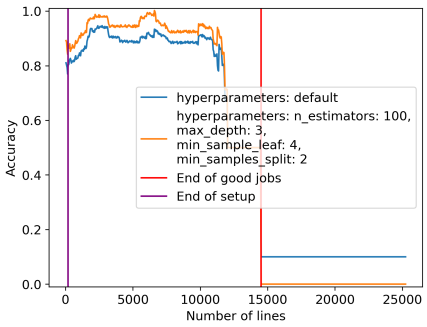
(c) Classifier: RF, Tokenizer: Word



(d) Classifier: GB, Tokenizer: Char



(e) Classifier: GB, Tokenizer: GPT4



(f) Classifier: GB, Tokenizer: Word

Figure 6.30: Best parameters for each combination of classifier and tokenizer, showing results of predictions on all datasets for the Lines model running on job 3 with equal classes.

For each of the subfigures in 6.30, Table 6.14 provides the DTW distance between the default and best hyperparameters for each of the combinations evaluated. The one that showed the most difference is 6.30e featuring a distance of 344.35 while all others have a distance between 1.32 and 3.41. Table 6.14 features the performance numbers when training the predictor on job 3 with the Lines model. It shows that the GB classifier takes a substantial more amount of time to train then RF as the GB classifier takes about 51 times longer to train.

Table 6.14: DTW distance between the best hyperparameters and the default hyperparameters as well as the performance impact on training with different classifiers, tokenizers and hyperparameters with the Lines model on job 3.

Classifier	Tokenizer	Balancing	N-estimators	Max depth	Min samples leaf	Min samples split	RAM usage (MB)	Time taken (s)	DTW distance
RF	Char	Equal	150	3	1	8	9434	240	2.45
RF	GPT4	Equal	150	None	1	4	4210	580	1.63
RF	Word	Equal	100	None	2	2	5159	890	1.32
GB	Char	Equal	100	3	2	8	9250	12359	3.41
GB	GPT4	Equal	100	None	1	2	4351	5305	344.35
GB	Word	Equal	100	3	4	2	5159	2357	1.72

6.3.2 Analysis

Balancing classes:
Data for balancing for equal classes, 70%/30% split and all samples used was provided in Figures 6.16, 6.17 and 6.18. Balancing the classes provided a great insight into how data balancing greatly affects the MCC, score and accuracy of the different models and classifiers got affected when compared to 100% of the data being used. The first observation is that RF in all scenarios performs better than GB and as

such there is no point in using GB over RF if the hyperparameter tuning does not show significant improvement for GB.

The second observation is that the instances where unsuccessful jobs had a prediction accuracy of 0% during the entire pipeline simulation now instead had at least some successful predictions. Although the accuracy got better in some cases such as when using RF in conjunction with the word tokenizer and 70% balancing the results were still abysmal. The word tokenizer shows throughout all the combinations tested that it almost always performs worse than the other two tokenizers, and as such there is no point in testing it in future iterations. There are instances where the word tokenizer is performing better than expected on unsuccessful jobs but then the predictions for successful jobs plummets. Such is the case with GB classifier and 50% balancing. In contrast, RF with the GPT4 tokenizer and 50% balancing has about the same accuracy for predicting unsuccessful jobs as the word tokenizer but the accuracy for successful jobs increases over time and is between 80% and 100% instead of hovering around 20% accuracy.

The third observation was that the jobs were not sometimes of the same length. The job was changed along the way, and as such, there were instances where the largest train job could be only 498 lines of logs while the test data featured 708 lines of logs. This can cause a problem as the classifier would not have any previous knowledge of such a job. It still got the prediction right as it probably understood that successful jobs tend to be longer than unsuccessful jobs. With this also comes the issue that shuffling in different ways can bring other results, especially if the jobs log structure has changed a lot in the previous months. One argument is then that the prediction should only be made on logs after the job log structure changed. But this can also be bad practice, as it would mean someone must keep track of when the log has changed so much that a prediction based on previous logs is invalid.

Moving away from the accuracy and instead focusing on the MCC score illustrated a clear downgrade. In almost all instances, except for with Figures 6.16b and 6.16e the MCC score got closer to 0 which indicates that the classifiers just guessed the correct classification label. There was however one instance in Figure 6.16a were although the MCC score got lower compared to using all samples it still successively climbed and the score increased the further along the job got.

Balancing showed that the accuracy got a lot worse and that the MCC score got closer to 0 in almost all instances. This is not a good result and can be caused by several, of which two of them are further discussed in the analysis on hyperparameter tuning on job 1 and on how job 2 and 3 perform with the models.

Hyperparameter tuning:
There are two angles to look at with Hyperparameter tuning. The first is how the various parameters tested affect the accuracy and the second angle is how big the performance impact it has. Performance differed greatly between running on default values as seen in Table 6 versus running on the best parameters for accuracy seen

in Table 9 on the GB classifier. The biggest performance increase of 72% when running GB with the char tokenizer is a great improvement. The other combinations of running the GPT4 and Word tokenizer also showed improvement but not as much as the char tokenizer. RF showed little to no difference in performance and the same can be said for accuracy as well.

When combining the RF classifier with different classifiers, it showed no real improvement on job 1 when measuring the accuracy. Only when running with balancing at equal classes did the accuracy improve, but for some reason it increased in the beginning of the build to fall sharply thereafter. This points to it at the start of the build being able to better classify for it to then fall in the middle of the CI job simulation and in the end it roses again.

Parameter tuning the GB classifier showed much greater improvement and the biggest gain was when running the char tokenizer in Figure 6.20j and equally balanced classes. Instead of it quickly falling to 0% accuracy in the end it instead sharply rose to 100% accuracy. Other combinations evaluated also showed good improvement when compared to RF. The big takeaway for the parameter tuning is that GB is more important to tune than RF when running log classifying tasks. Moving forward in future iterations the parameters that had the best accuracy will be used.

In conclusion, the Hyperparameter tuning did not change any result that dramatically that it completely would change the conclusion for the thesis. Even though it did help certain combinations a lot such as 6.20j, there are still combinations that provide a better result even with no real change after the Hyperparameter tuning, such as 6.20g and 6.20l.

Lines model:
For job 1, the Lines model proved to be less reliable than the Logtime model, as its prediction accuracy fluctuates more significantly. This inconsistency might mislead developers into believing that, at a certain point in the build process, the algorithm is highly confident that the build will fail, prompting them to investigate potential failure causes only to find none. During the next prediction after it could then predict that the build is going to pass the CI job.

One potential solution to improve the Lines model could be to consider not only the current prediction but also the previous two predictions. If any of these predictions indicate a successful build, the model would refrain from notifying developers of a failure. However, this approach has potential drawbacks. For instance, if there are never three consecutive predictions indicating failure, the model may become ineffective. Additionally, this method could delay the information reaching developers, as more predictions are needed before issuing a warning about an unsuccessful build.

When comparing the performance between the Logtime and Lines model when both use the best hyperparameters, there is a clear trend when comparing the two. The

results from Table 6.9 and 6.10 show that the Lines model takes significantly longer to train than the Logtime model, this while the RAM usage is almost as high. This is because the RAM usage reported is the peak during the training session, even so a large discrepancy was expected as the lines model only stores the data from up until the line it is training on meanwhile the Logtime model needs all data at all time while training. The amount of time taken by the Lines model compared to Logtime is expected because the Logtime model only needs to train once while the Lines model trains 71 times for job 1. It should also be noted that the hyperparameters play a huge part in how long it takes to train. One example of this is the GB classifier and Word tokenizer that took 1008 seconds to train on the Lines model compared to 13 seconds on the Logtime model. This can be compared to the same classifier, GB, being used in conjunction with the GPT4 tokenizer taking the same amount of time for the Logitime model as with the word tokenizer. Meanwhile, the Lines model took 1/3 less time training with the GPT4 contra word tokenizer on the GB classifier.

Testing on different jobs:
Testing on different jobs have shown that the value of the predictor can be based on what type of job it is predicting on. Job 2 showed in Figure 6.25 that it behaves in the same way as job 1, meaning that the prediction remains largely stable throughout the job and at around 80% accuracy depending on what tokenizer and classifier used. Therefore the, data on job 2 does not change the analysis in any way compared to job 1. However, the MCC scores for job 2 was slightly lower than expected when comparing to the accuracy. With an accuracy of 80% it would be expected that the MCC score should be at least 0.6 but in reality it was at the best 0.5 in Figures 6.25d and 6.25f.

Job 3 is more interesting in that it behaves completely differently than job 2. Although the accuracy in the beginning and middle of the job are as expected: Around the point of 12000 lines something happens that makes the accuracy fall down to 50%. When all unsuccessful jobs are still running the accuracy drops to 0%. This job in particular shows why using MCC is so important as it shows the RF classifier is only predicting on one class correctly in hence why the accuracy is so high as 90% in Figure 6.27a. A score of 0 in this case with a high accuracy would indicate that all successful jobs get classified correctly but almost all unsuccessful jobs gets classified incorrectly. This is really odd behavior as in this case both the best hyperparameters are used as well as balanced classes.

Testing on multiple jobs have shown that the type of job is of upmost importance when deciding whenever to implement a near-real-time build outcome prediction. Both job 1 and 2 have shown great result in both accuracy and MCC metrics. Meanwhile job 3 has shown terrible performance in MCC score.

6.4 Iteration 4

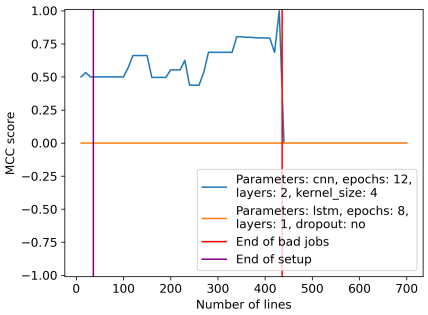
Previous iterations have worked with implementing traditional ML classifiers for CI build outcome prediction. This iteration instead focuses on DL and the classifiers CNN and LSTM. The testing will encompass both seeing the effect on how many layers are best suited for each classifier as well as the number of epochs that provide the best results. All evaluated combinations can be found in Table 6.15. Only balancing on equal classes are used for iteration 4. The Table shows what combinations are evaluated on both CNN and LSTM. As the architecture on both are different so parameters are not able/needed to be tuned by both, one such parameter is that the kernal size for CNN has been tested with 2, 3 and 4. In the case of LSTM the addition of a Dropout layer has been tested. As the LSTM classifier is much more complex and with the results in previous iterations showing a tendency to overfit, it was decided to also evaluate the LSTM model with and without a Dropout layer. For all other parameters used on job 1, GridSearchCV was used and for the other two jobs RandomSearchCV was used, where half of all possible combinations was evaluated. The reasoning behind the decision of using such low number of epochs such as 4 is due to the previous iterations overfitting on the data. The more the DL algorithm learn from the data the higher the possibility to overfit. The same reasoning is also behind the decision to use layers 1, 2 and 3 as adding more layers adds more complexity to the classifier.

Table 6.15: Combinations evaluated in the fourth iteration.

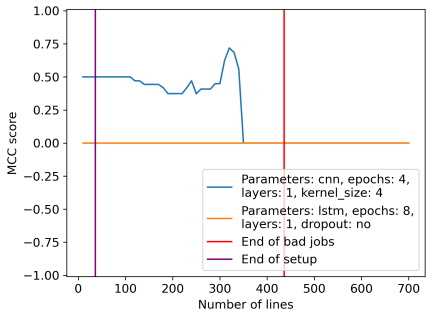
Jobs:	Job 1
	Job 2
	Job 3
Classifiers:	CNN
	LSTM
Tokenizers:	Char
	GPT4
	Word
Models:	Logtime
	Lines
Kernel size:	2
	3
	4
Layer count:	1
	2
	3
Epoch count:	4
	8
	12

6.4.1 Results

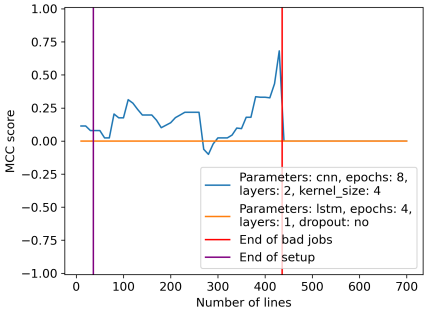
This chapter will focus on comparing the two deep learning classifiers CNN and LSTM. The only combination that will be shown for each of them is the one which performed the best in terms of accuracy. All combinations have been tested using equal classes and with removed timestamps. First tested combinations is with job 1 which is displayed in Figures 6.31 and 6.32 were the two classifiers were very close in terms of accuracy but not in terms of MCC. The MCC score of the CNN classifier consistently remained higher than the LSTM classifier, with a few exceptions: in Figures 6.31c between lines 280 to 300, Figure 6.31e at line 260 and lastly Figure 6.31f at line 300. Conversely, the LSTM classifier generally stayed at a score of 0, except for a deviation to 0.25 in Figure 6.31d at line 80, and two other minor deviations of 0.05. This consistent 0 score for the LSTM classifier suggests that it either overfits or underfits the data. Although the CNN classifier usually scored above 0, its performance was heavily influenced by the tokenizer used, except for the instances where it underperformed compared to the LSTM. The best performer for being consistent in MCC score was in Figure 6.31a.



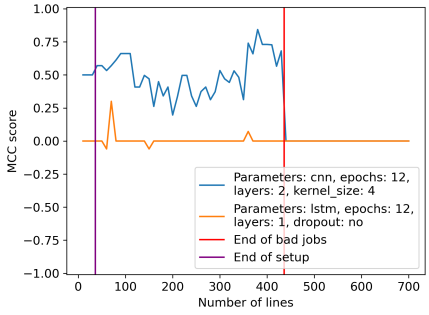
(a) Model: Logtime, Tokenizer: Char.



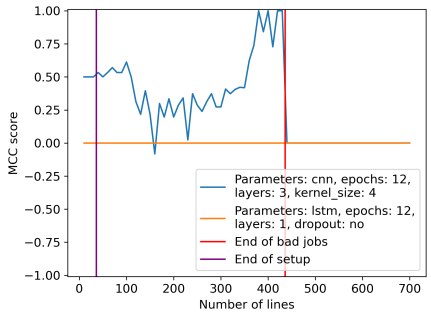
(b) Model: Logtime, Tokenizer: GPT4.



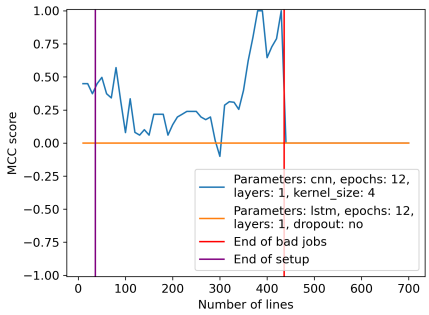
(c) Model: Logtime, Tokenizer: Word.



(d) Model: Lines, Tokenizer: Char.



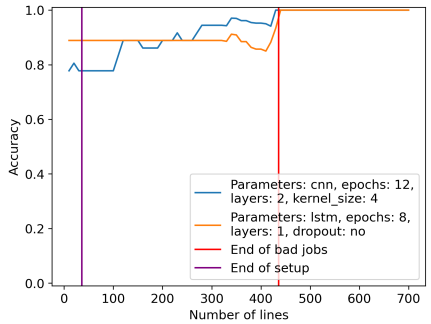
(e) Model: Lines, Tokenizer: GPT4.



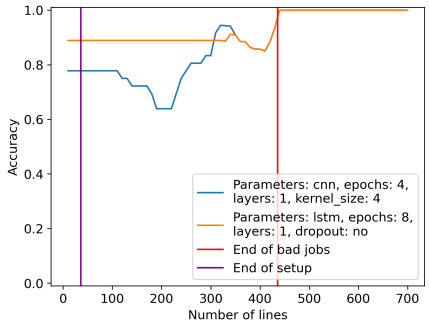
(f) Model: Lines, Tokenizer: Word.

Figure 6.31: Best layer and epoch count for each combination of classifier and tokenizer, showing results of predictions for the Logtime and Lines models running on job 1 with equal classes using MCC as measurement.

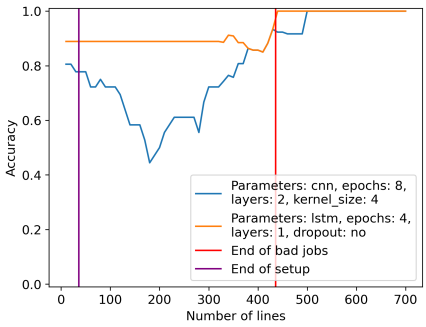
Accuracy for the Logtime and Lines model with the different DL classifiers is shown in Figure 6.32 for job 1. The result from the Logtime shows that the LSTM classifier performs identically on all three tokenizers in Figures 6.32a, 6.32b and 6.32c. The accuracy was still stable for most of the CI build process at slightly below 90%. Only at line 340 did the accuracy slightly increase for it to decrease to 80% accuracy at line 400, afterwards it increased again to 100% as all unsuccessful jobs had ended. The CNN classifier got more affected by the tokenizer used where the char tokenizer displaying clearly better accuracy than the word tokenizer and for the most part the GPT4 tokenizer. In the Lines model the LSTM classifier displayed the same characteristics as with the Logtime model with one key difference in that the accuracy dropped to 0% at line 620 in Figure 6.32e. In Figure 6.32d two similar drops can be noticed but the accuracy here dropped to about 15% instead of 0%.



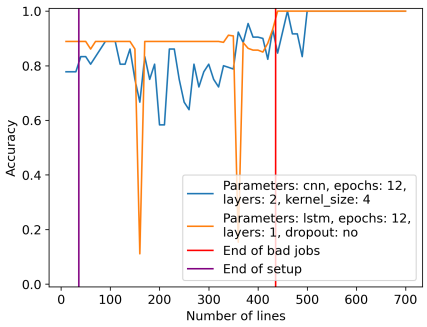
(a) Model: Logtime, Tokenizer: Char.



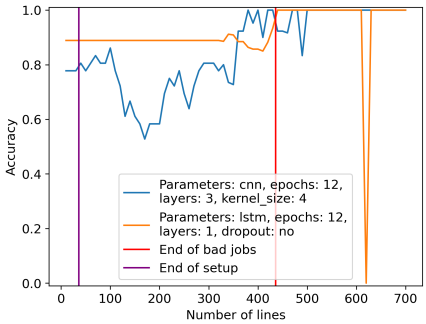
(b) Model: Logtime, Tokenizer: GPT4.



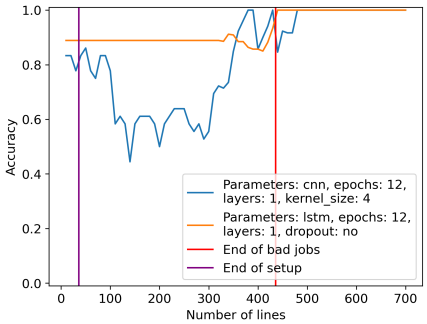
(c) Model: Logtime, Tokenizer: Word.



(d) Model: Lines, Tokenizer: Char.



(e) Model: Lines, Tokenizer: GPT4.



(f) Model: Lines, Tokenizer: Word.

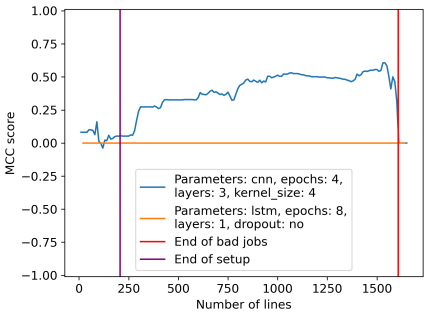
Figure 6.32: Best layer and epoch count for each combination of classifier and tokenizer, showing results of predictions for the Logtime and Lines models running on job 1 with equal classes using accuracy as measurement.

The computational power needed for the Logtime model was much lower than for the Lines model in Table 6.16. The longest training session was the LSTM classifier and that did not take even one minute, in contrast for the Lines model the lowest training time was just over 10 minutes, indicating a more than 10 times increase in time needed to train. The largest time difference was a 38 times increase for the LSTM model with the word tokenizer, when comparing the Lines and Logtime models. The difference in computational power needed between the tokenizer was not as massive as between the models. In all cases the GPT4 tokenizer was the easiest to train with the word tokenizer always taking the longest. Time increased in between the fastest and slowest tokenizer was always less than 100%.

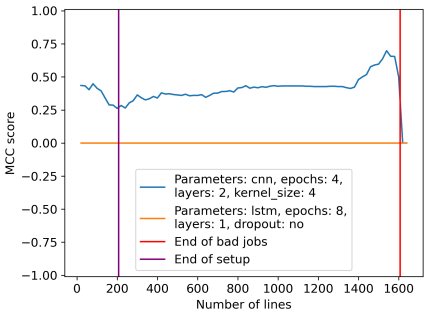
Table 6.16: Performance impact on training with different DL classifiers, models, tokenizers and best hyperparameters on job 1.

Classifier	Model	Tokenizer	Epochs	Layers	Time taken (s)
CNN	Logtime	Char	12	2	23
CNN	Logtime	GPT4	4	1	19
CNN	Logtime	Word	8	2	46
LSTM	Logtime	Char	8	1	52
LSTM	Logtime	GPT4	8	1	25
LSTM	Logtime	Word	4	1	44
CNN	Lines	Char	12	2	644
CNN	Lines	GPT4	12	1	603
CNN	Lines	Word	12	3	1353
LSTM	Lines	Char	12	1	1457
LSTM	Lines	GPT4	12	1	1055
LSTM	Lines	Word	12	1	1674

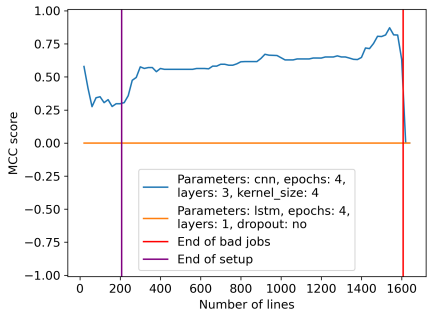
The LSTM classifier showed the same pattern in Figure 6.33 as was already seen in Figure 6.32. The CNN classifier however showed more favorable result, in Figure 6.33a the MCC score almost continuously increased throughout the CI build run. At certain points there where dips in the MCC score such as at line 100, line 750 and between lines 1,000 to 1,300. But overall the MCC score increased from 0.1 to 0.3 from line 240 to 260, it had then increased from 0.3 to about 0.4 at line 770 and at line 1100 it had reached 0.5 for it to peak at 0.6 right before the unsuccessful jobs had ended. In Figure 6.33b the MCC score begins at around 0.25 right after setup is completed, increasing slightly thereafter to 0.35 and for the rest of the job until line 1,350 it keep making minuscule increases to the MCC score. After line 1,350 it increases more sharply to reach 0.6 at line 1,500, for it to thereafter drop towards a score of 0 as all of the unsuccessful jobs are completed. The MCC score seen in Figure 6.33d was after line 350 higher than the other two combinations featuring the Logtime model. The MCC score was consistently between 0.55 and 0.6 throughout the building process and like the other two combinations, the MCC score increased at line 1350 for it to fall back to 0 after line 1,500 right before all unsuccessful jobs had ended.



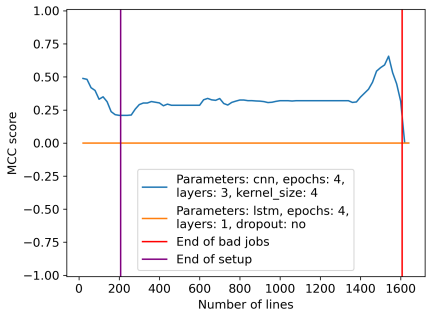
(a) Model: Logtime, Tokenizer: Char



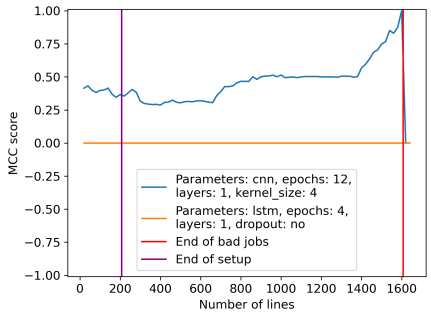
(b) Model: Logtime, Tokenizer: GPT4



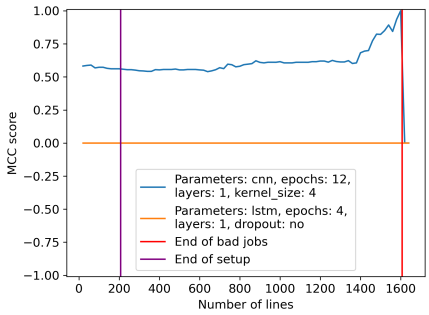
(c) Model: Logtime, Tokenizer: Word



(d) Model: Lines, Tokenizer: Char



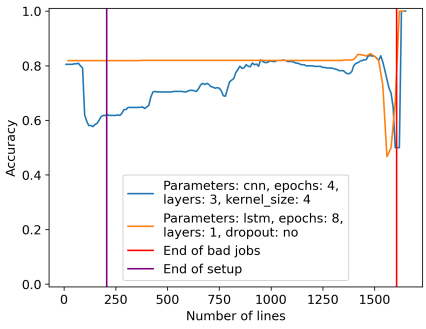
(e) Model: Lines, Tokenizer: GPT4



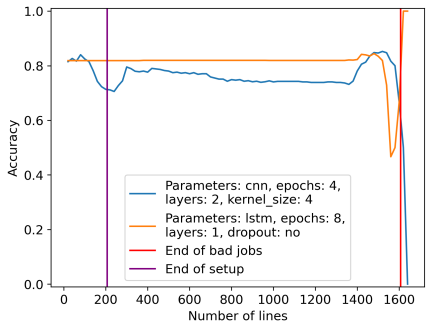
(f) Model: Lines, Tokenizer: Word

Figure 6.33: Best layer and epoch count for each combination of classifier and tokenizer, showing results of predictions for the Logtime and Lines models running on job 2 with equal classes using MCC as measurement.

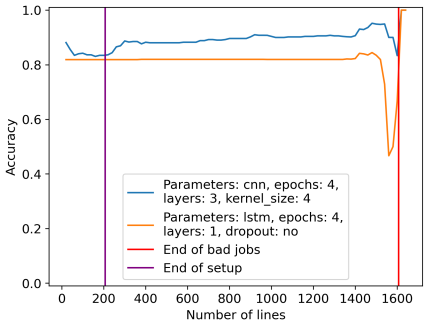
Figure 6.34 illustrates that the LSTM classifier performed identically when paired with the Logtime model. The CNN classifier was however performing differently depending on the tokenizer used where the char tokenizer and GPT4 tokenizer in Figures 6.34a and 6.34b for most of the CI build performed worse than the LSTM classifier. Figure 6.34a showed an accuracy of 0.65 right as the setup was complete, this accuracy then increased in at certain lines where it otherwise stayed at the exact same accuracy in between. These points can be seen at lines 270, 400 and lastly at 650. After a small decrease from line 700 to 750 the accuracy increased to stay slightly below 80%. This combination has a similar decrease in accuracy as the LSTM classifier at line 1,530, but for this combination it occurred slightly later at line 1,600 instead. At an accuracy above 80% for the entire CI build Figure 6.34c had the best combination for job 2 in terms of accuracy. Apart from before setup and 100 lines thereafter, the accuracy never fell below 85% and continued to increase as the build progressed. It also did not see a sharp decrease in accuracy at line 1,500 as the other combinations evaluated did. The accuracy did however decrease by 10% points.



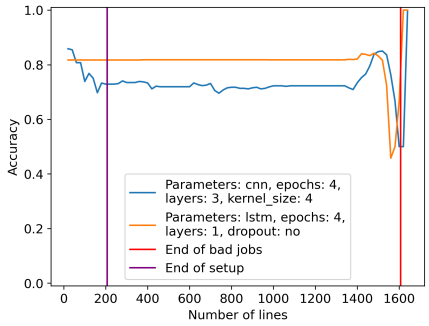
(a) Model: Logtime, Tokenizer: Char.



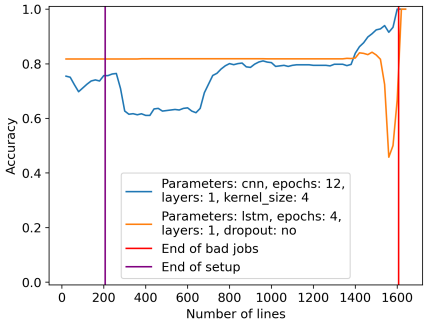
(b) Model: Logtime, Tokenizer: GPT4



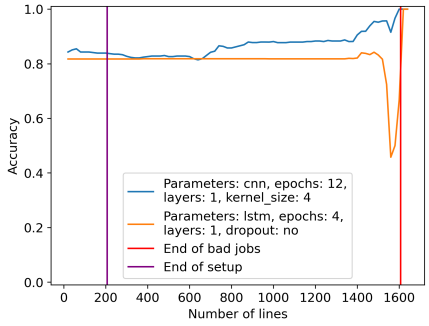
(c) Model: Logtime, Tokenizer: Word.



(d) Model: Lines, Tokenizer: Char.



(e) Model: Lines, Tokenizer: GPT4.



(f) Model: Lines, Tokenizer: Word.

Figure 6.34: Best layer and epoch count for each combination of classifier and tokenizer, showing results of predictions for the Lines model running on job 2 with equal classes.

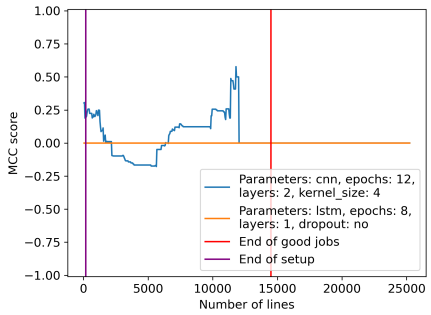
Table 6.17 provides information on how long time it took to train the different combinations used on job 2. The table shows that the Lines model is considerably slower to use for training the classifiers and that this increase can be as much as almost 30 times, with the lowest increase being close to 15 times.

Table 6.17: Performance impact on training with different DL classifiers, models, tokenizers and best hyperparameters on job 2.

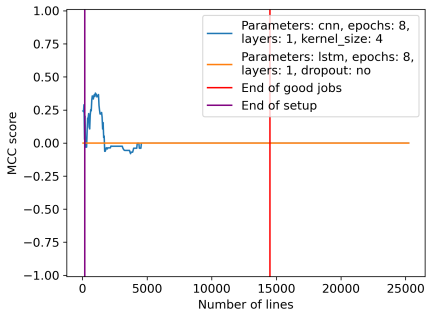
Classifier	Model	Tokenizer	Epochs	Layers	Time taken (s)
CNN	Logtime	Char	4	3	608
CNN	Logtime	GPT4	4	2	511
CNN	Logtime	Word	4	3	1362
LSTM	Logtime	Char	8	1	1289
LSTM	Logtime	GPT4	8	1	993
LSTM	Logtime	Word	4	1	1398
CNN	Lines	Char	4	3	17859
CNN	Lines	GPT4	12	1	15317
CNN	Lines	Word	12	1	20888
LSTM	Lines	Char	4	1	23612
LSTM	Lines	GPT4	4	1	20128
LSTM	Lines	Word	4	1	25350

For the Logtime model the LSTM classifier showed the same pattern as previous jobs where the MCC score stayed at 0. For the CNN classifier with the char tokenizer in Figure 6.35a the score starts at 0.25 and stays around there for 1,000 lines. After which the score goes down to -0.25 meaning that there is a disagreement between the predictions made and the observation. The MCC score does however increase again above 0 but at around 13,000 lines it reaches 0 again and stays at 0 for the build time left. Figure 6.35a provided a similar result in the beginning of the build resulting in score above 0 at first but between lines 1,000 and 5,000 the score was negative. Thereafter the score stayed at 0 for the rest of the builds. The overall best result from the Logtime model was seen in Figure 6.35c, where the score never was below 0. For the first 5,000 lines the MCC score hovered around 0.25 where fluctuations can be seen. There is a downward trend in the score from line 4,500 to line 7,000 where a large drop can be seen thereafter. After the MCC score had stayed at slightly above 0 up until line 10,000, at that point the score first increased to 0.20 and thereafter to 1 something unique to this combination.

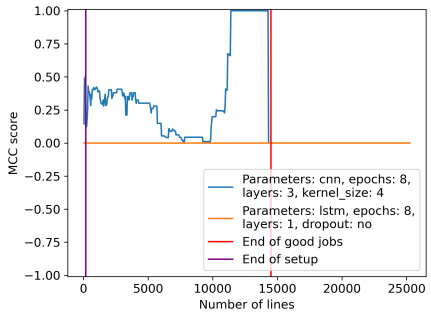
The Lines model performed for the most part worse than the Logtime model, this is seen in Figure 6.36d where the Lines model, between lines 12,500 and 15,000 had a score of -1. If however looking at the first 6,000 lines of the build it scored higher instead. Overall it was still worse since the drop to -1 indicates that all samples are predicted on wrongly. On the other hand the Lines model with the GPT4 tokenizer got a higher MCC score overall in Figure 6.36d compared to the Logtime model in Figure 6.35a. The contributing factor was at the beginning right after the setup had completed where the Lines model did not drop at all in the MCC score. Lastly, the word tokenizer in Figure 6.35f suffered from the same thing as the char tokenizer with the Lines model, namely that it did not receive the same increase to a score of 1 at line 12,000. Apart from that it never had a higher score than the Logtime model.



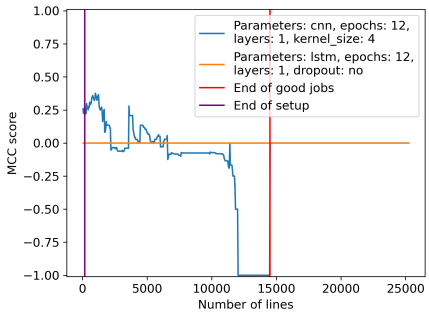
(a) Model: Logtime, Tokenizer: Char.



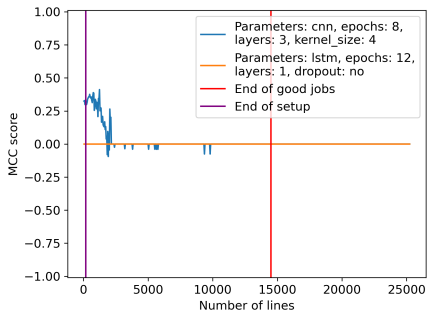
(b) Model: Logtime, Tokenizer: GPT4.



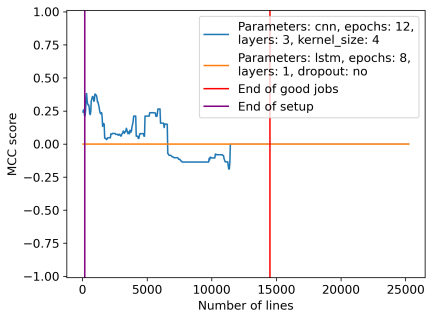
(c) Model: Logtime, Tokenizer: Word.



(d) Model: Lines, Tokenizer: Char.



(e) Model: Lines, Tokenizer: GPT4.



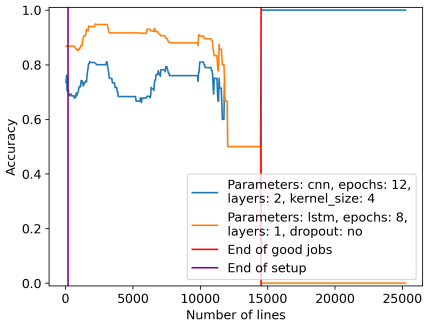
(f) Model: Lines, Tokenizer: Word.

Figure 6.35: Best layer and epoch count for each combination of classifier and tokenizer, showing results of predictions for the Logtime and Lines models running on job 3 with equal classes using MCC as measurement.

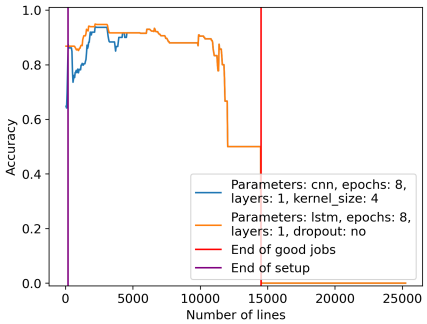
Accuracy measurements for the same combinations as in Figure 6.35 can be found in Figure 6.36. The first result in Figure 6.36a displays that accuracy goes back and forth. After setup the accuracy is at 70%, then it increases to 80% and thereafter falls back again at line 3,000. After line 6,000 it increases again to 80% but after line 1300 it increases to 100% and thereafter, when all successful builds have finished it falls to 0%. The GPT4 tokenizer in Figure 6.36b follows the LSTM classifier after line 6,000 has been reached, beforehand it showed and accuracy lower than the LSTM classifier. Figure 6.36b also showed less accuracy compared to all LSTM combinations, however it also showed that a combination can reach 100% at around

line 13,000. It was at 100% accuracy for the rest of the build duration's, with one exception being at right before the end of the last successful builds where it quickly dropped to 0% accuracy.

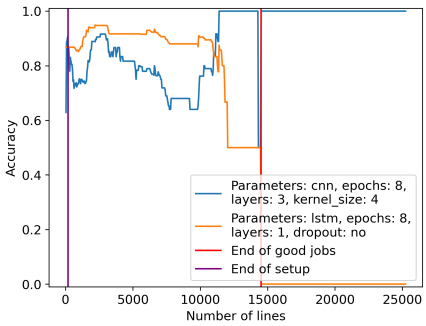
The first Lines model displayed in Figure 6.36d got better accuracy overall up until line 13,000 compared to the Logtime model in 6.36a. Apart from one larger accuracy decrease seen at line 5,000 which reached 15% points lower compared to the Logtime model, the accuracy was in the 3,000 lines before and 5,000 lines after around 10% points better. However, at near the end of successful jobs between lines 13,000 and 15,000 the accuracy dropped to 0%. The same drop in accuracy was not seen in the other two combinations containing the Lines model, namely in Figures 6.36e and 6.36f.



(a) Model: Logtime, Tokenizer: Char.

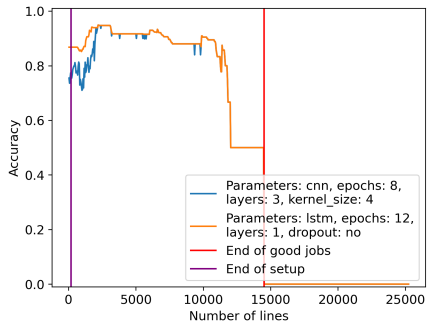


(b) Model: Logtime, Tokenizer: GPT4.

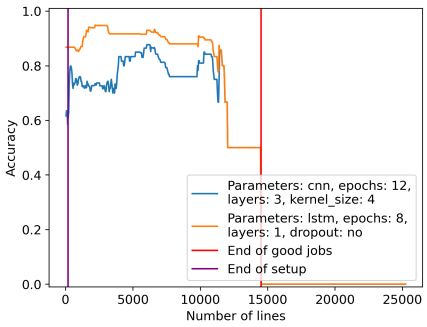


(c) Model: Logtime, Tokenizer: Word.

(d) Model: Lines, Tokenizer: Char.



(e) Model: Lines, Tokenizer: GPT4.



(f) Model: Lines,Tokenizer: Word.

Figure 6.36: Best layer and epoch count for each combination of classifier and tokenizer, showing results of predictions for the Logtime and Lines models running on job 3 with equal classes.

To get a better overview of what it takes to train job 3, Table 6.18 provides the metrics for the best combination of classifier, model and tokenizer in conjunction with the best set of epochs and layers. The table shows that training with the Logtime model takes less than 1 hour meanwhile the Lines model takes more than 10 hours. This makes it take more than 10 times as long to train a DL classifier with the Lines model.

Table 6.18: Performance impact on training with different DL classifiers, models, tokenizers and best hyperparameters on job 3.

Classifier	Model	Tokenizer	Epochs	Layers	Time taken (s)
CNN	Logtime	Char	12	2	999
CNN	Logtime	GPT4	8	1	803
CNN	Logtime	Word	8	3	2655
LSTM	Logtime	Char	8	1	1847
LSTM	Logtime	GPT4	8	1	1677
LSTM	Logtime	Word	8	1	2741
CNN	Lines	Char	12	2	32696
CNN	Lines	GPT4	12	1	28542
CNN	Lines	Word	12	3	44880
LSTM	Lines	Char	12	1	45960
LSTM	Lines	GPT4	12	1	31014
LSTM	Lines	Word	12	1	55963

6.4.2 Analysis

The results have shown that CNN is in this case superior to LSTM in terms of MCC score, found in Figures 6.31, 6.33 and 6.35. Even though the LSTM model was more fine-tuned during this iteration, the CNN model outperformed it. All Figures in this iteration featuring the MCC metric have shown that it was almost impossible to get

the LSTM model to reach a MCC score higher than 0. The Lines model showed in for example 6.31d that it was possible to get the LSTM classifier combinations to provide a score apart from 0. As the Lines model contains less data than the Logtime model, one possible answer as to why the LSTM classifier overfitted was its complexity in conjunction with the complexity of the logtime model.

There can be several reasons why CNN is better in this scenario and this can be seen in several figures presented in this iteration, one example being Figure 6.32c where LSTM has a tendency to overfit the data as its a more complex classifier compared to CNN. This is also the most likely cause as to why CNN outperformed LSTM, as can be seen by all Figures 6.32, 6.34 and 6.36 the LSTM model with 1 layer was always better or the same as using two or three layers instead. This strongly suggests overfitting. For the CNN model this was never the case and Table 6.16 shows that even on job 1, the classifier is sometimes better with either two or three layers compared to one. For the other two jobs, the amount of data is more than 70 times more than for job 1 as can be seen in Table 6.8. Due to the increased availability of data, the CNN classifier performed the best when paired with a tokenizer and an optimal layer count suited to that tokenizer. Meanwhile LSTM was still performing the best using one layer. As the LSTM classifier is more complex it typically needs more data in order to not overfit, the more data available for each sample can explain the behavior.

When using the Lines model in combination with the char tokenizer, the LSTM classifier also performed better in Figure 6.36d. The reason why is the same as with the Logtime model. Although the Lines model has less data overall, the char tokenizer features more data points than the other tokenizers as every character is a data point rather than a segment of characters as is the case with the other two tokenizers.

In order to mitigate the overfitting problem with LSTM it was evaluated how adding a Dropout to the model would affect the accuracy/MCC score. The answer was that it did not affect the final result at all, this is also why all figures feature "dropout: no". As the Dropout did not affect the accuracy/MCC score of the LSTM classifier lead to believe that the features chosen for the models need to be changed in order to better utilize LSTM.

Furthermore, the LSTM classifier also took longer to train compared to the CNN classifier. The most probable answer is because LSTM is more complex and as such it is harder to train such a classifier. The biggest contributing factor to the training time is the number of epochs used and to a lesser degree the number of layers used. However, even when comparing the same number of epochs and layers such as for job 3 with the Lines model and the GPT4 tokenizer, the CNN classifier was still faster by about 9%. The overall takeaway from this iteration is that CNN is better than LSTM for the usecase of the study. Both in terms of accuracy and computational resources used. The final three models that are best suited and could be explored further are RF, GB and CNN.

6.5 Iteration 5

All prior iterations have focused on building an ML algorithm, but in order to best use such an algorithm, it has to be able to provide useful feedback to the developer. The interview was divided into two different segments were the first looked at how feedback is best brought back to developers, where multiple mockups was shown. The second part included the prediction algorithm and what thresholds it needs to achieve in order to be viable. The structure of the result follows the same structure as the interview template which can be found in Table A.1.

6.5.1 Results

The first part of the interview asked questions related to RQ 1.3 which aims to find out how feedback is best presented back to the developer. For the first question also question number 5 in the interview guide at Table A.1, interviewees with ID1, ID3 and ID6 use Slackbots for getting back information from the CI system today. They all said the configure the bots to send notifications in Slack once a commit has either gotten a verified mark inside Gerrit where the mark can either say the commit passed CI or it failed CI. Other interviewees with ID2, ID4 and ID5 all said they do not use the Slackbots, giving almost the same reason of getting to many notifications. Where interviewee ID2 said "It gets too noisy to listen to them" and the same reason was also given by ID4 where they instead follow the commit in real-time instead by looking at how what stage it has reached in the integration process. Meanwhile ID5 giving the same reason but instead keeps taps on commits by regularly checking Gerrit for an overview of which commits have passed/failed CI.

Later when asking how the feedback was used and how it affected their work almost everyone answered in the same way. How interviewee ID1 said it sums up what most of them said "If I have pushed something and get a notification that it fails I usually try to check it immediately as soon as I can". People also gave different remarks on what usually happens for different types of commits or what they do should they not have the time right away. ID1 keeps sticky notes whenever it is not possible to get to it right away. ID6 meanwhile lets lesser critical patches lay until there is a time gap where not much is happening, meanwhile mission-critical commits get looked at as soon as possible.

For question 5b everybody had the same opinion namely, with one quote from ID1 being "I think the current feedback system is good enough, I cannot think of a better way.". There were however some improvements mentioned by ID6 which talked a lot about refinement that other companies have already implemented. One such suggestion is that different test failures in the CI log can be automatically marked red by the CI system instead of the normal flow of the developer having to search through the log for that CI stage. Another improvement mentioned was that related test failures in CI could automatically be sent back to Gerrit in the form of a comment so developers can avoid going to the CI log at all. ID2 also expressed a desire to have everything in one place, as currently the company utilizes so-called

"rigs" which are computers that are supposed to sit inside the car. The problem is that all rigs need their own log system and the logs can therefore become scattered in different places.

The interview moved on to show the different figures and what participants saw as the best solution. No one saw any use for developers for the first solution in question 6a. Comments like "I today newer follow my commits but instead wait for a notification on whenever it went wrong or succeeded. So I don't see myself using such a system" from ID1 and the following from ID3 "For easy jobs, I would just wait for the jobs to pass, but would probably not go out of my way to look at it regardless.". There were however some people such as ID5 and ID6 who saw some use for it by the people tuning the algorithm as well as CI developers. ID6 had the following to say "I do not think I would have sat and followed these predictions, but still think it might be good to have the graphs for CI developers who can tweak the algorithm." and the following from ID5 "I would personally never use it but can see it potentially useful for people developing the machine learning algorithm". The overall consensus from all participants was that this was the worst solution of the ones presented.

The participants were more favorable towards the option in question 6b, with participants ID2, ID3, ID4 and ID6 saying they would use it, albeit with a few caveats. ID1 and ID5 both said they would not use it and both stated the same reason for the quote from ID1 "The issue I have with this is that if the predictor can see that this build is going to fail, should I not as well be able to see that the job has failed.", this is implying that if the predictor can figure out that the CI build is going to fail, it should have already warned the developer that something has gone wrong. For the participants saying they would use it, ID2 said would gladly tweak the thresholds in order to find the correct prediction level. Both ID3 and ID4 said they would use it but would want the people who develop the algorithm to set the threshold, alternatively the people developing CI. Then later on the developer can tweak it the his liking. interviewee ID6 also liked the idea especially that the predictor would point out in the log on what it based its failure prediction. Adding the comment "It is better to get this log directly in Gerrit instead of putting it in the CI system". The interview template featured different questions for Gerrit (6b) and a Slackbot (6c), but the results were the same for both questions with the only difference is that people already using the Slackbot (ID2, ID4 and ID5) preferred it while others preferred Gerrit (ID1, ID3 and ID6).

All interviewees looked favorable on the third solution discussed in question 6d, but where opinions were divided was on what context needed to be added. ID1 and ID5 both stated more context needed to be added but did not have any suggestion on how to do it, only stating something like ID5 said "why does it usually fail". Interviewees ID2, ID3 and ID4 all suggested that it can be used for seeing what tests are likely to fail. ID2 saying the following "Definitely would like, if it can also tell that the following tests have a high chance of failure that would be a big plus. This could also provide feedback to testers about where extra gunpowder needs to

be added". ID3 for the most part stated the same, ID5 however added that this should be a part of a pre-commit/pre-push hook, stating the following "If you have tests that can run in less than 1 second, you should instead show a list of those tests. Otherwise, it would be better to have it as part of pre-push or pre-commit. One worry is that already pre-push takes too long which causes developers to avoid it". ID6 apart from what others had already mentioned added the following "This would also have been interesting from a CI point of view as well as one could look for which files give the most technical debt with which tests fail".

If the predictions were to be shown in the IDE most developers did not have a preference between in the sidebar or the bottom, as was the case with interviewee ID4, ID5 and ID6. meanwhile ID1 and ID2 preferred to have an icon in the sidebar stating that the shown icon was larger than the small text at the bottom. ID2 said the following "Better in the sidebar, bigger icon than in the bottom but red color is better." ID3 however preferred at the bottom as it is used to notifications being at the bottom and the prediction was seen as a notification in this case. All participants also agreed that constant blinking was annoying but there was a disagreement between only changing the color, all preferred red in this case, or having it blink only for a couple of seconds. interviewee ID2 and ID5 preferred blinking for a couple of seconds were ID2 gave the following statement "If it keeps blinking I would not want to ignore clicking on it as it would get annoying. With only a few seconds I would notice its there but not feel the urge to click it". The others preferred the button just changing color, ID6 stating the following "Would prefer the button to be red but not flashing, this is comparable to Slack turning red and catching my attention".

There were mixed feelings on whether the prompt showing after clicking the prediction button should be on the file level, line level or function level. Interview ID3 was the only person which did not like the approach of only having a button you have to click, instead also wanting the function to be underlined in a color like how it is usually underlined in red today if something is wrong. Also stating that the color should not be red but a different color in this case as to not indicate failure but instead indicate that there can be a failure. For the button however, everyone agreed that red was the best color. Everyone also agreed that having the prediction on the file level is a good approach, with participants ID4, ID5 and ID6 all voicing concern that files that are too long might frustrate the developer more than help in this scenario. Therefore, in this case ID5 said the following "If a file is let's say 3,000 lines I would have no idea where to look in that file. If it is narrowed down to about 300-400 lines that would be more useful".

Most of the participants had already at this point answered questions 7, 7a and 7b therefore the interview continued with how developers perceived the work of continuously making predictions when running inside of a pipeline. ID1 stated that 80% accuracy was required but also noted the previous point on human resources being more expensive than computer resources "like spending an extra hour trying to figure out why is this prediction, saying that this might fail instead of just waiting for CI. High risk with bad predictions". ID2 meanwhile also took the time aspect into

account stating the following "Gradually decreasing value, must have more security at the end here it is 90% that applies. At the beginning, a worse prediction of 80% can be ok when there is like 15 minutes left". ID3 said that his workflow consists of patches that are almost certain to pass all checks, this would need more than 95% to actually look at it. For patches more prone to failures 75% is enough. ID4 set a baseline at 80% as a first test but added that this should probably be closer to 90%, while ID5 said 90% upfront. The one outlier to the other interviewees is ID6 which suggested A/B testing and then having the values 70% and 90% accuracy thresholds to see how people react to different values. It was also added that if a value has to be given it would be 90%.

When question 8a was posed everyone said that they have to test the algorithm in the real world before coming up with an answer. There were however some who thought about it but no noteworthy answers were given apart from what was stated in question 8.

Everyone said that they prefer a lower false-positive rate than a false-negative rate, ID3 even retracted the previous statement of 75% being enough now instead saying the "75% accuracy is with false-positives". When later on asked why this was more important all again had the same response of "it causing less headache for developers" as was said by ID5. What was implied was that if the developer gets notified and the developer has to spend time looking at successful builds, the developer would get frustrated.

Lastly, when asked how much time would be saved ID6 kept insisting on doing an A/B testing approach to see how the developer using it felt, also noting that "time does not matter as long as it provides perceived productivity for the developer". ID5 said 5 minutes at the least, stating that it would otherwise not be worth spending the time when the build is so close to getting the real result. Both ID3 and ID4 said that the prediction system should run until the build has finished, both also stated that their reason for this is because of hanging jobs. If it would be possible for the predictor to catch hanging jobs this can save multiple hours as it would not have to wait for the timeout of the job to be reached. As previously stated ID2 mentioned that it has to save about 15 minutes. ID1 was the most skeptical adding that it would have to save at least 30 minutes in order to weigh in that it can sometimes be wrong in making its predictions.

6.5.2 Analysis

The results on how feedback today is reported at Zenseact shows a mixture of using Slackbots, Gerrit as well as following the commit inside of a CI system. These are all preferences and shows that in order to implement such a solution in industry, several apps have to be accounted for. As the second picture shown (Figures A.4 and A.6) looked more favorable, one approach could be to both send a message in Gerrit as well as a Slack message which leads to the same spot for taking in the data, as is shown by the web link in the figures.

Getting answers on how developers use the feedback today could provide a way to get data on whether they would use the prediction system without explicitly asking. Developers who answered that they look at the feedback could be more inclined to actually use the prediction system. While developers who only look at the feedback when there is time to spare would be less inclined to actually use such a system. The two outliers who today tried to immediately check feedback on patch sets which were not as favorably inclined towards the prediction system were ID1 and ID5. This shows that at least 2/3 of all interviewees would be inclined to use such a prediction system.

Providing feedback to developers
When it came on how to use the CI build outcome predictor all respondents favored the second approach compared to the first one, with setting up a threshold and sending a notification when the threshold has been reached. However, some respondents namely ID3 and ID4 wanted CI developers to first set the threshold before using it. This means 2/3 of all interviewees wanted to configure it themselves. The best solution in this case would be for CI developers to first set up the threshold and then let developers tweak it to their liking. The one issue with this is that as some participants talked about, different jobs might require different thresholds that are dependent on time saved as well as the resources required to run the CI job. As the CI job might change in the future this may add extra work onto CI developers. Therefore the solution that satisfies everyone may not be the best solution depending on the different factors.

The first approach with the link to a diagram showing the results of predictions could still be useful for e.g. CI as ID5 and ID6 pointed out. Although this would not be intended for developers an assumption can be made that it is unnecessary to provide the link inside Gerrit as it would clutter up the comment feed. Instead it can exist as an artifact inside the CI system so that CI developers can have access to it, this would also allow regular developers to see it as well. As no CI developers were interviewed it is impossible to say if this would be the best solution or even if they would gain any useful information from such a diagram. Alas this question is left unanswered as it was not in the scope of the thesis to see how CI developers could maintain a prediction tool.

The third option set out to evaluate the prediction system of the thesis against how other prediction systems have been used as explained more in Chapter 3. The results show that all participants preferred the approach used by other researchers but instead of checking the probability of the CI build getting a failure, interviewees were keener on what test cases are more prone to failure. As for how to report the test failures there were two different solutions mentioned, one was to get it in the idea, one was to run short tests in pre-push/pre-commit. Without another interview explicitly asking about what way is better, it is hard to draw any conclusions on how to implement test case prediction as these were ideas obtained from the interview.

One other aspect of getting details in the IDE was that it could tell the developer to be extra careful. All developers liked the approach of having it in the IDE more so than the solution to have a threshold. On how such a solution could be integrated all interviewees disliked the idea of constant blinking and some preferred blinking for a couple of seconds while others would it rather just turn red instead. As such a first solution could feature a button at the bottom of the IDE with the text changing red whenever the prediction system wants the developer's attention, with there is also an option to have a blinking effect for a couple of seconds for those who prefer it.

The conclusions drawn are that the preferred option for developers is to have a prediction system integrated into their IDE and that it gives the developer feedback on which tests are likely to fail given the edited lines/file. The second most preferred option is to give feedback on which file is likely to cause a failure inside CI, although with this approach it can be at max 400 lines that the developer has to look at otherwise more context is needed such as what function to look at. The third best option is for build outcome prediction to reside inside the CI system where the developers can set thresholds for when the predictor thinks CI is likely to fail.

Perceived usefulness of just-in-time build outcome prediction
The lowest accuracy anyone answered was 75% and the highest was 90%. There were also some who answered 80% and with the data obtained the accuracy would have to be at least 80% in order for at least half of the respondents to use a Ci build outcome prediction system. As interviewee ID6 mentioned a good approach would be to begin with A/B testing with several developers to get further data on what developers think vs how they act as that can be entirely different. The answers also showed that question 8a was superfluous as no relevant data was gained.

All participants mentioning that a higher false-positive rate is better shows that testing on both successful builds and unsuccessful builds separately was the right choice. In the final analysis, the accuracy on successful builds is prioritized over unsuccessful ones.

Half of the interviewees said that the prediction system should run all the way until the build has finished. Meanwhile other developers believed a time limit should be set up. There is some value in letting the predictor run all the way in order to only catch hanging jobs, otherwise a time limit should be set up. As such the proposed solution is to have a limit of 5 min before the build ends, with it being adjustable by the developer who uploads the commit. The predictor also being aware of the potential for a hanging build which means it informs the developer even if the time limit has passed.

The results have shown that the perceived usefulness of just-in-time build outcome prediction is good. The accuracy should at least be 80% and that is for successful builds, although the accuracy on unsuccessful builds is also important, it is more important to spare human resources compared to computer resources. Especially when predictions can be made using normal personal computers compared to tests

which sometimes run on specialized rigs that are harder to come by. The time saved is also important, with at least 15 minutes saved should be aimed for.

7
Discussion

This chapter provides a thorough discussion of the study's outcomes. It starts by systematically addressing each research question; it also offers a structured approach to dissecting the findings. Moreover, the chapter includes a section dedicated to threats to validity which demonstrates a commitment to transparency and rigor in the research process. By acknowledging potential limitations upfront, the study's credibility is bolstered, and avenues for improvement are identified in a future work section.

RQ1.1: To what degree is it possible to predict the result of a CI job in just-in-time based on previous data from the same pipeline?
The findings reveal that the artifact exhibits an MCC score always surpassing 0, indicating it does not merely "guess" the build outcome. This is true for all three jobs when using the best combinations, as can be seen in Figure 6.31a for job 1, Figure 6.24f for job 2 and Figure 6.26e for job 3. One concern is that the prediction accuracy does not significantly improve over time, indicating that new log data fails to enhance accuracy as expected. In Figures 6.32a, 6.25f and 6.27e which display accuracy instead of the MCC score, the expected outcome would have been for the accuracy to start at around 50% and then steadily rise towards 100% as only successful builds remain. Theoretically, as the predictor receives more data, it should be able to identify more distinct patterns that differentiate successful from unsuccessful jobs.

Pinpointing the exact cause for the accuracy staying largely the same is challenging, but it is likely due to logs being mostly similar between successful and unsuccessful jobs until failure occurs. Although if this was the case the accuracy of predicting should be around 50% whereas now, depending on the combination used it typically starts between 60% and 90% accuracy. Depending on the job setup, the failure either terminates the job immediately or proceeds if tests are ongoing. This is why in most cases there is an increase in accuracy at the end of the job as the number of unsuccessful builds gets lower.

To better understand which features impact the result, the effect of removing versus retaining the timestamp for job 1 was compared. This comparison showed minimal impact on the overall result. One other preprocessing step involved removing binary output, which mostly consisted of error outliers in the dataset, though its impact on the final result was not investigated. However, these outliers can affect the result, particularly when real text output is replaced with binary output, as occurred in

certain samples in the log.

In conclusion, whilejust-in-time prediction of CI job outcomes is feasible, the simulation alone in this thesis is not sufficient to ascertain its adequacy for developers. With only a small subset of combinations tested it is still not possible with 100% certainty to say that the just-in-time build outcome prediction used in this thesis is more feasible than regular CI build prediction that is performed before the job has started running.

RQ1.2: What is the relationship between the computational power needed to predict the build outcome in just-in-time and the size of the job parameters?
The results show that job 1, the smallest job only requires at max 270 MB of RAM and takes no longer than 30 seconds to train, seen in Table 6.9, when using equal classes and the Logtime model. The Lines model shows similar RAM usage but with much higher training time taken at 1008 seconds, displayed in Table 6.10. This should be put in contrast to the fact that the dataset as a whole is really small with a size of 2.6 MB with equal classes. With how powerful computers has become today this should be of no concern, as was also stated by some interviewees who argued that saving human work hours is more beneficial than saving computer resources. It should however be noted that the larger jobs 2 and 3 takes upwards of 10000 MB, which is a huge increase compared to 270 MB of job 1. Job 2 can take upwards of 14000 seconds to train while job 3 takes 12000 seconds to train on the Lines model, seen in Table 6.13 and Table 6.14. These numbers are true for both traditional ML as well as DL.

When comparing job 2 and 3 it can in Table 6.8 be seen that each log sample in job 3 is larger than the logs in job 2. As noted earlier, job 3 takes longer time to train than job 2. The conclusion that can be drawn is that larger samples impact the training time more than the number of samples used. The RAM usage scales more around the same as both job 2 and job 3 used around the same amount of RAM.

Overall, training is expensive however it is not so expensive that it is a reason to not use the prediction system. Often at night the CI system is mostly sitting idle compared to the day and as such it would be an opportune time to train such a model. Even during the weekend where it would be possible to run the training for longer.

Looking at the different jobs and how they scale, it can be seen that they scale very irregularly where it can in some instances be almost linearly to in the worst case almost exponentially. Furthermore, it can also be seen that the more log data that is used for training, the amount of time taken to train will grow almost linearly to in the worst case below exponentially. This means that when choosing to use the prediction algorithm on larger datasets something has to be done to optimize the workflow. This can for example either be to down sample the amount of data available for training or to reduce hyperparameter tuning, although as shown in the

thesis this can cause huge swings in accuracy/MCC score. Another possibility would be to use a more powerful CPU/GPU which can handle the training faster. Despite the significant size difference between job 1 and the other jobs, there are jobs with even more available data. For instance, one observed but not selected job, referred to as job 4, has a size of 4000MB when using equal classes. Using the same scale between job 1 and job 2 as reference. This could in theory mean that the RAM usage could be more than 50,000MB and that the time taken would be over 100,000 seconds for training one combination. The practical issue could then be that one would need to choose between down-sampling, reducing hyperparameter tuning or to increase the computational resources available. Either way this could have an huge impact on the end result.

Performance impact when doing predictions is also not an issue provided enough resources is given to the prediction system, at max each prediction takes 3 seconds and as the entire dataset is not loaded into memory the RAM requirements are substantially lower than for training. The one issue is that enough resources need to be put to the prediction system such that it is possible to accommodate all jobs currently running, which should be predicted upon. If each prediction takes 3 seconds, this would mean that a queue can accumulate if enough resources are not put into the prediction system, thus making it fall behind the CI system. By falling behind the CI system is that there can be one job which has two unprocessed predictions in the queue.

Overall an exponential growth in time taken to train the more log data available can be seen. It is also necessary to not only focus on the training part but also to ensure that there is enough compute power to ensure data does not end up in a queue.

RQ1.3: How can the feedback on the most probable failure cause from the just-in-time prediction best be displayed to the developer?
The feedback from just-in-time prediction is best displayed by setting up a default threshold for different parameters and the letting the developers themselves change these as needed. Once the threshold is met a notification system sends out a notification notifying the developer that the CI job is predicted to fail. This lead to a place where feedback on what has caused the predictor to predict the outcome can be seen.

The study shows that developers prefer to be notified rather than to themselves having to actively follow predictions. Although the preferred way for developers at Zenseact was to use an already existing set up and integrate it into that feedback system. Developers did not have a single opinion as to how predictions should be displayed and as such there were people who would rather have it directly inside of Gerrit while others thought it was okay having it as an artifact in the CI system.

Developers have different preferences and this shows that. As for the best place, the question has to be left unanswered, but it is possible to answer the question of providing a summary which most developers was positive towards. This leads to the conclusion that the best way to provide feedback is through a notification system

which sends a link to where a summary of why the CI build is predicted to fail can be found.

RQ1.4: What is the perceived usefulness of just-in-time build outcome predictions for software developers?
Developers are overall positive towards predicting the CI build outcome, but some remain skeptical on how it can actually work until it is fully implemented and used. Developers saw more interest in trying approaches that uses the result of a CI pipeline in mapping which tests fail to what files are edited. This way it would be possible to tell developers when they edit the file or are about to commit it that there are some tests that they can run first before uploading it to Gerrit.

Even though the approach of predicting test failures was mentioned by almost all developer in the interview, it does not say they did not see any use from the CI build outcome prediction system. Almost all developers were in favor as a way to reduce the human workload as the workflow could be less disruptive as it would result in not having to wait for a full CI job run. By predicting the likely cause of a CI failure, developers can receive targeted feedback on potential issues. As illustrated in Figure A.5, this approach highlights only the latest prediction made by the system, enabling developers to quickly identify the relevant failure without sifting through the entire CI log.

Some developers were skeptical on how the parameters available in a CI job could help predict the build outcome as they argued that the CI job should already be aware of this fact that there is an error that has occurred. There is merit to what they say and this is also confirmed with the results. However, in the CI builds there are also edge cases that is frustrating to encounter. One such issue brought up by developers was how the prediction system could help with warning should the build become in a hanging state. Another is when there is a failure in the setup that is not discovered, this failure then leads to other code that will fail. The log will show both failures but the first failure may be missed as the developer usually expect just one failure.

The conclusion is that the perceived usefulness of just-in-time build outcome predictions for software developers is overall positive but is a case where it has to be seen working in practice before a final judgement can be given.

RQ1.5: At what point in the build process is it no longer useful to continue making predictions?
Developers tends to think differently about the question as was shown by the interview. Most however tended to agree that when the build only has about 15 min left is a good stopping point. Some also mentioned 5 minutes as a good stopping point, with one person mentioning 30 minutes.

When factoring in at what point to stop making predictions it is also important to factor in that the longer the build has run, the larger the log and thus the re-

quirement for RAM goes up. It also frees up CPU resources as well although as the CPU is not utilized as much as the RAM, it is not as big of a factor whilst being a factor. This was also brought up by some interviewees, but a consensus was that human resources was more important and that it should therefore be the sole factor when deciding on when to stop predicting.

During the interview some developers mentioned such a prediction system can also be good for checking for hanging CI builds and could therefore still be useful to make predictions all the way through until the job is successful. In order to see if the build was hanging the prediction systems parameters would have to be altered compared to how it is now. Although prediction on such a case would be beneficial, other results would be altered as the parameters would change.

The conclusion is therefore that 15 minutes before the build has finished is when we should stop making more predictions. The conclusion has been made as the resources required should mostly be ignored and that most developers want it to be at 15 minutes. An option for developers wanting 5 minutes or another time frame is to be able to set the setting themselves with a default at 15 minutes. A default would be set by either the CI team or the team responsible for the prediction system, as was suggested in the interview. A default value should be used as some developers wanted a default set for it to then be adjustable.

RQ2: To what extent can the problems be solved by the potential solutions in (RQ1)?
The solution presented in this thesis can help solve the problem of reducing the time taken for CI jobs. In the best case scenario, it can achieve an accuracy of 90% and a MCC score of 0.75 in job 1. Depending on the job that is used for build prediction this figure can vary, as was the case in job 2 where the best result was accuracy of 80% and a MCC score of 0.5 with the score starting to climb near the end of the job. Job 3 did not perform as well as the other two jobs did with a MCC score of 0, which is not acceptable. The accuracy was however stellar at 90%, but overall not the best result, when comparing this work to what others have found this is around the same score as has been achieved by other studies when predicting the build outcome before the integration process has begun. This shows that the problem can be solved with the solution presented in this thesis, but the earlier the feedback can be given the better.

Overall the most time that can be saved when looking at build outcome prediction before the CI job has started is for job 1 3500 seconds, but more realistically around 700 seconds if the outliers are removed, this can be seen in Figure 6.4. For job 2 around 4500 seconds can be saved, this is displayed in Figure 6.12 and for job 3 in Figure 6.14, 2000 seconds can be saved realistically. This is based on where most jobs are ending on the time scale, but rounded up to accommodate some outliers. Since the predictor can usually only predict with enough accuracy when the job is about to fail, usually due to it spotting a test error, the time saved is very minimal at around 1 minute.

More time can however be saved as the log summary given to the developer will be very concise and thus possibly easier for the developer to understand than the full log. This is of course if the predictor is making the right assumption on when the build will fail. Without the use of developers actually using the system in production it is currently impossible to say how much time is saved with using the prediction system. It is also possible that developers would need to take time to look at wrong predictions as well and this would take time. So in order to be able to make such a calculation on time saved one would need to first take the time saved compared to a full CI build and add that with the time saved by only having to read the concise log compared to a full CI log. The time saved would then need to be subtracted with the time taken to look at wrong predictions. With this one would get the time saved with the prediction system.

Other studies examining build outcome prediction algorithms have not clearly demonstrated the optimal implementation methods. This thesis concludes that the best approach largely depends on the individual developer. Noting that it should feature some kind of notification system, how this is to be implemented is what differs between developer. All seem positive to gain the feedback from the same feedback system used today for alert on when a CI job has failed, this can be either through the use of email or a bot in a messaging service. When receiving the notification and after having looked at the cause of possible build failure, the developer can then decide whenever to terminate the build or to let it running.

Another option would be to directly terminate the build if the build is predicted to fail, there are however several issues with this. The issue with this is the increase in the human workload compared to the computer workload. With an accuracy of 80%, this would mean 1 in 5 builds would be wrongfully be terminated. In the interview it was found that computational power is almost always less expensive than human work hours. If the build is then terminated wrongfully, the developer would then have to first retrigger the build and wait for the build to complete, which will be longer than just waiting out the previous build. This would further disrupt the workflow for the developer, one problem the prediction system is trying to solve.

The most preferred way for developers to use predictions was through the use of linking tests failed in CI to what files has been edited in a commit. This information could then be given to the developer through their IDE which could help avoid failures in CI in the first place. Although the CI build prediction tool does not currently support this feature, it is a step of the way to implement such a functionality as it already downloads the necessary data.

7.1 Threats to validity

Internal validity
As with all qualitative studies, bias in data analysis is always a threat. In order to mitigate these problems multiple approaches was used when analyzing the data,

which included Time Series Analysis as well as Hypothesis testing. Furthermore, during the analysis of the results from each iteration as well as the planning for the next iteration the inclusion of both supervisors in the discussion helped widen the scope of the analysis.

Another point of concern is the code that was produced. As with all software, bugs can occur, and this is true as well with this project. Often internal functions have safeguards for common mistakes, but even so there are no safeguards to ascertain that data is handled correctly. It is impossible to completely eliminate the problem, but it is possible to mitigate it with the help of testing as well as careful planning and adequate quality assurance. Whenever there was a library available for use, this code is usually more tested by several experienced developers. Furthermore, the confined range of hyperparameter tuning poses another possible threat, as it was restricted by computational constraints. These limitations might have resulted in less than optimal model performance.

External validity
As the study was conducted on 3 different jobs which compile and build software intended for use with cars, the results may not be generalizable on other pipeline jobs in other industries where the software is built differently. Even if multiple pipeline jobs are tested in order to avoid bias, it is not feasible to test for all jobs as the time to validate each job would take a considerable amount of time. Each CI system has its own way of presenting and writing logs. Therefore, some parts of the data processing may be specific to how the log data is presented by the logs from these 3 jobs from one company.

Moreover, the thesis used a simulator in order to simulate already finished CI pipeline builds. This means the tools have not been run in a real pipeline setting, in practice however it should be easy to implement the tool in a pipeline. One approach would be that the pipeline sends streams of the log data as it is written to the CI job log.

Using interviewees with a software developer background for interviews may influence the results as each developer will have a different experience using the CI system and thus different views. The interview will however ask about how long they have worked with the system and how they view their knowledge of it. But as it is humans who are interviewed it may be biased toward how the developer thinks. All developers are also affiliated with a single company and will thus be confined to how the company is structured. To combat this all interviewees chosen had previous experience from other companies in order for the result to be more generalizable.

7.2 Future work

Predicting the outcome of CI builds in just-in-time represents a novel approach with ample room for improvement. While initial efforts have focused on leveraging CI logs for prediction, there remains untapped potential in exploring alternative data sources and methodologies. A critical avenue for enhancement involves examining

additional metadata available during CI builds beyond what has been considered. It would also be possible to further improve the result by looking at certain features in the log and how it affects the predicted outcome. One such feature could have been to look at the output type. By incorporating this supplementary information, the aim is to refine prediction models and elevate their accuracy.

Furthermore, future studies could involve the implementation of a Proof of concept within a CI system, followed by rigorous A/B testing to evaluate various thresholds and feedback mechanisms. This empirical approach would provide valuable insights into developers' preferences and perceptions regarding prediction accuracy and usability.

An integral aspect of the future research involves soliciting feedback directly from developers to guide the efforts. Through interviews, the goal is to ascertain developers' perspectives on predictive models and identify their most pressing needs. One recurrent request from developers has been the integration of CI job outcomes with test failure analysis, particularly linking failed tests to the corresponding edited files. This feedback underscores the importance of providing actionable insights within developers' IDEs or through pre-push/pre-commit hooks. By alerting developers to potential test failures prior to CI execution, the aim is to streamline development workflows and enhance overall efficiency.

8

Conclusion

This study's aim was to improve the CI feedback loops through the use of build prediction, as well as how feedback is best presented. There are currently multiple approaches presented to predict the CI build outcome before the job has started but few studies look into how such a prediction system can best be implemented into the CI feedback loop. Furthermore, in order to improve the accuracy of predictions, this thesis sought to produce an artifact that utilizes parameters during the CI job runtime. To assess the findings, interviews with developers were conducted to gather their perspectives on predictions and on what the most effective ways to integrate them into their workflow.

The research revealed that developers are overall positive about the use of predictions to improve the feedback loop. With near-real-time prediction, there were a variety of challenges that needed to be resolved. The result of the research is in the form of an artifact that acts as a way on how to best implement a just-in-time CI prediction system and how to best incorporate it into a CI feedback loop for developers.

The artifact provides a guide for how a similar system can be implemented and later evaluated on a CI system by anyone. The discoveries outlined in the thesis offer valuable guidance for professionals across different fields on overcoming the challenges associated with deploying and using just-in-time prediction systems. Furthermore, it provides researchers with knowledge on how to next proceed in the research regarding just-in-time CI build outcome predictions. In conclusion, the study contributes to the expanding knowledge base on CI build outcome prediction by providing a fresh perspective on the use in near-real-time.

Bibliography

[1] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681, 2013.

[2] T. A. Ghaleb, D. A. da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Software Engineering*, vol. 24, pp. 2102–2139, Aug 2019.

[3] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 197–207, 08 2017.

[4] S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *2013 Agile Conference*, pp. 121–128, 2013.

[5] G. Pinto, M. Reboucas, and F. Castor, "Inadequate testing, time pressure, and (over) confidence: A tale of continuous integration users," in *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 74–77, 2017.

[6] B. Chen, L. Chen, C. Zhang, and X. Peng, "Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 42–53, 2020.

[7] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at google)," 05 2014.

[8] K. W. Al-Sabbagh, M. Staron, and R. Hebig, "Predicting build outcomes in continuous integration using textual analysis of source code commits," *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022.

[9] G. Pinto, F. Castor, R. Bonifacio, and M. Rebouças, "Work practices and challenges in continuous integration: A survey with travis ci users," *Software: Practice and Experience*, vol. 48, no. 12, pp. 2223–2236, 2018.

[10] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards automated log parsing for large-scale log data analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 931–944, 2018.

[11] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," *Empirical Software Engineering*, vol. 25, pp. 1095–1135, Mar 2020.

[12] F. Cannizzo, R. Clutton, and R. Ramesh, "Pushing the boundaries of testing and continuous integration," in *Agile 2008 Conference*, pp. 501–505, 2008.

[13] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of java-based open-source software," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 345–355, 2017.

[14] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 41–50, 2014.

[15] H. Wang, C. Ma, and L. Zhou, "A brief review of machine learning and its application," in *2009 International Conference on Information Engineering and Computer Science*, pp. 1–4, 2009.

[16] C. V. Gonzalez Zelaya, "Towards explaining the effects of data preprocessing on machine learning," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 2086–2090, 2019.

[17] A. Subasi, *Practical Machine Learning for Data Analysis Using Python*. 06 2020.

[18] R. Mohammed, J. Rawashdeh, and M. Abdullah, "Machine learning with oversampling and undersampling techniques: Overview study and experimental results," in *2020 11th International Conference on Information and Communication Systems (ICICS)*, pp. 243–248, 2020.

[19] H. Aguinis, R. K. Gottfredson, and H. Joo, "Best-practice recommendations for defining, identifying, and handling outliers," *Organizational Research Methods*, vol. 16, no. 2, pp. 270–301, 2013.

[20] M. Fares, S. Oepen, and Y. Zhang, "Machine learning for high-quality tokenization replicating variable tokenization schemes," in *Computational Linguistics and Intelligent Text Processing* (A. Gelbukh, ed.), (Berlin, Heidelberg), pp. 231–244, Springer Berlin Heidelberg, 2013.

[21] `https://www.nltk.org/api/nltk.tokenize.TreebankWordTokenizer.html` [Accessed: 2024-03-28].

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[23] A. Mustapha, M. Lachgar, and K. Ali, *An Overview of Gradient Descent Algorithm Optimization in Machine Learning: Application in the Ophthalmology Field*, pp. 349–359. 06 2020.

[24] X. Ying, "An overview of overfitting and its solutions," *Journal of Physics: Conference Series*, vol. 1168, p. 022022, feb 2019.

[25] D. Berrar, *Cross-Validation*. Academic Press, 2019.

[26] S. Scott and S. Matwin, "Feature engineering for text classification," in *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, (San Francisco, CA, USA), p. 379–388, Morgan Kaufmann Publishers Inc., 1999.

[27] M. Feurer and F. Hutter, "Hyperparameter optimization," *Automated machine learning: Methods, systems, challenges*, pp. 3–33, 2019.

[28] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 ed., 2009.

[29] N. Seliya, T. M. Khoshgoftaar, and J. Van Hulse, "A study on the relationships of classifier performance metrics," in *2009 21st IEEE International Conference on Tools with Artificial Intelligence*, pp. 59–66, 2009.

[30] S. J. Al'Aref, K. Anchouche, G. Singh, P. J. Slomka, K. K. Kolli, A. Kumar, M. Pandey, G. Maliakal, A. R. van Rosendael, A. N. Beecy, D. S. Berman, J. Leipsic, K. Nieman, D. Andreini, G. Pontone, U. J. Schoepf, L. J. Shaw, H.-J. Chang, J. Narula, J. J. Bax, Y. Guan, and J. K. Min, "Clinical applications of machine learning in cardiovascular disease and its relevance to cardiac imaging," *European Heart Journal*, vol. 40, pp. 1975–1986, 07 2018.

[31] D. Cutler, T. Edwards, K. Beard, A. Cutler, K. Hess, J. Gibson, and J. Lawler, "Random forests for classification in ecology," *Ecology*, vol. 88, pp. 2783–92, 12 2007.

[32] N. Jalal, A. Mehmood, G. S. Choi, and I. Ashraf, "A novel improved random forest for text classification using feature ranking and optimal number of trees," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 6, Part A, pp. 2733–2742, 2022.

[33] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in Neurorobotics*, vol. 7, 2013.

[34] C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz, "A comparative analysis of gradient boosting algorithms," *Artificial Intelligence Review*, vol. 54, pp. 1937–1967, Mar 2021.

[35] X.-W. Chen and X. Lin, "Big data deep learning: Challenges and perspectives," *IEEE Access*, vol. 2, pp. 514–525, 2014.

[36] R. Qamar and B. Zardari, "Artificial neural networks: An overview," *Mesopotamian Journal of Computer Science*, vol. 2023, pp. 130–139, 08 2023.

[37] B. J. Wythoff, "Backpropagation neural networks: A tutorial," *Chemometrics and Intelligent Laboratory Systems*, vol. 18, no. 2, pp. 115–155, 1993.

[38] A. F. Agarap, "Deep learning using rectified linear units (relu)," 2019.

[39] Z. Hu, J. Zhang, and Y. Ge, "Handling vanishing gradient problem using artificial derivative," *IEEE Access*, vol. 9, pp. 22371–22377, 2021.

[40] S. Narayan, "The generalized sigmoid activation function: Competitive supervised learning," *Information Sciences*, vol. 99, no. 1, pp. 69–82, 1997.

[41] Y. Kim, "Convolutional neural networks for sentence classification," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 08 2014.

[42] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6, 2017.

[43] C. C. Aggarwal, *Training Deep Neural Networks*. Springer International Publishing, 2018.

[44] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[45] I. Saidani, A. Ouni, M. Chouchen, and M. W. Mkaouer, "Predicting continuous integration build failures using evolutionary search," *Information and Software Technology*, vol. 128, p. 106392, 2020.

[46] J. Xia and Y. Li, "Could we predict the result of a continuous integration build? an empirical study," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 311–315, 2017.

[47] E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson, and M. Castell, "Supporting continuous integration by code-churn based test selection," in *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, pp. 19–25, 2015.

[48] M. Staron, M. Ochodek, W. Meding, O. Söder, and E. Rosenberg, *Machine Learning to Support Code Reviews in Continuous Integration*, ch. Chapter 6, pp. 141–167. 2021.

[49] I. Saidani, A. Ouni, and M. W. Mkaouer, "Improving the prediction of continuous integration build failures using deep learning," *Automated Software Engineering*, vol. 29, 05 2022.

[50] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting anomaly in big data system logs using convolutional neural network," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pp. 151–158, 2018.

[51] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[52] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," vol. 30, 07 2005.

[53] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pp. 81–90, 2006.

[54] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.

[55] Y. Fan, X. Xia, D. A. da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by szz on just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1559–1586, 2021.

[56] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), p. 966–969, Association for Computing Machinery, 2015.

[57] C. Khanan, W. Luewichana, K. Pruktharathikoon, J. Jiarpakdee, C. Tantithamthavorn, M. Choetkiertikul, C. Ragkhitwetsagul, and T. Sunetnanta, "Jitbot: An explainable just-in-time defect prediction bot," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1336–1339, 2020.

[58] M. Kawalerowicz and L. Madeyski, "Jaskier: A supporting software tool for continuous build outcome prediction practice," in *Advances and Trends in Artificial*

Intelligence. From Theory to Practice (H. Fujita, A. Selamat, J. C.-W. Lin, and M. Ali, eds.), (Cham), pp. 426–438, Springer International Publishing, 2021.

[59] J. v. Brocke, A. Hevner, and A. Maedche, *Introduction to Design Science Research*, pp. 1–13. Springer International Publishing, 09 2020.

[60] E. Knauss, "Constructive master's thesis work in industry: Guidelines for applying design science research," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 110–121, 2021.

A

Interview Template

Table A.1: Interview questions.

	General questions
1:	Are you okay with your voice being recorded?
1a:	Informed consent: The only people who may have access to the recordings are me, and my supervisors from Zenseact and Chalmers. The recordings will be deleted at the latest when the final report is submitted but no later than June 1st. If you later ask that the recording be deleted, it will be deleted at that time. All data will be anonymized, meaning no names or personal details will be shared.
2:	What is your role in the company?
3:	How long have you worked at Zenseact?
4:	Have you had any previous work experience with developing CI solutions?
4a:	If yes, how many years?
5	Explain the goal of the thesis and what it is trying to achieve.
	How can the feedback on the most probable failure cause from the just-in-time prediction best be displayed to the developer?
5:	How is feedback reported back to you today from the CI system?
5a:	In which way does it affect your work, do you ignore the feedback, or do you look at the feedback right away?
5b:	Would you say this is a good feedback system, why or why not? Is it possible to integrate the prediction algorithm this way?
6:	Show examples of how the prediction algorithm can be integrated in certain stages.
6a:	Inside Gerrit, link to Zuul artifact where predictions are displayed so it is possible to follow in near-real-time as predictions are made shown in Figure A.1. i.e. optional with no notification after the first link. The artifact that will be shown can be seen in Figure A.2 and the updated version once a new predicition has been made can be seen in Figure A.3. Do you see this as a good option?
6b:	Inside Gerrit, Once the commit has passed a certain condition on for example accuracy, number of lines in a job log etc. a message is sent to Gerrit, the message with link can be seen in Figure A.4. with the link to where in the log it thinks the failure is located, the CI log the has red text to mark were the failure is located, illustrated in Figure A.5.

6c:	With the use of a bot sending a message to a messaging service, Once the commit has passed a certain condition on for example accuracy, number of lines in a job log etc. a message is sent to the user, the message with link can be seen in Figure A.6. with the link to where in the log it thinks the failure is located, the CI log the has red text to mark were the failure is located, illustrated in Figure A.5.
6d:	Inside code editor, the system will see that a certain file/line tends to appear a lot when job is failing, indicating that the developer should be extra careful when committing this file. Is there a point in pointing out that the developer has to be extra careful with a certain file or line, based on earlier predictions? Is this best shown to developer with the text changing color, as can be seen when comparing Figures A.7 and A.8. Later a popup is shown as displayed in Figure A.9. Will blinking text be better to grab attention or be annoying? Would you say it is based on how fast the blink is? Is it better to have it in the sidebar instead of at the bottom of the IDE as illustrated in Figure A.10, the message will still be the same as Figure A.9?
7:	Can you think of a better way to integrate such a system into code editors?
7a:	Of the different ways what do you think is the best way?
7b:	Why is it better than the other ways?
	What does the perceived usefulness of just-in-time build outcome predictions for software developers?
8:	Now moving onto how you think about a prediction system. How accurate would such a prediction algorithm need to be to used in the company?
8a:	Why is x percent so important, why not x-5% or x+5% instead?
9:	When you say x percent, is this only for successful or unsuccessful jobs or for both jobs? Let's say successful jobs have x accuracy but unsuccessful jobs have a lower accuracy would that still be good to satisfy your requirements? Now instead lets say unsuccessful jobs have x accuracy but successful jobs have a lower accuracy would that still be good to satisfy your requirements?
9a:	Why are successful/unsuccessful builds getting categorized correctly more important?
10:	How much time in the CI system would you say must be saved for the solution to be worth it, ignoring the amount of system resources needed, factoring in it reaches the accuracy you mentioned earlier? At what point in the process would you say that it is better to wait out the CI job instead of making more predictions?
10a:	Why would you rather wait at that point?
	Other
11:	Do you have any other questions?

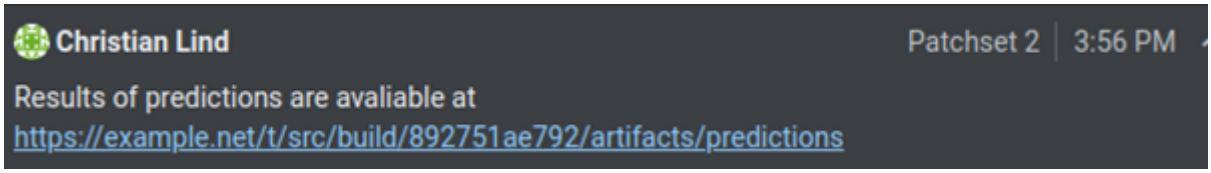


Figure A.1: Link to CI system inside Gerrit commit comment leading to artifact.

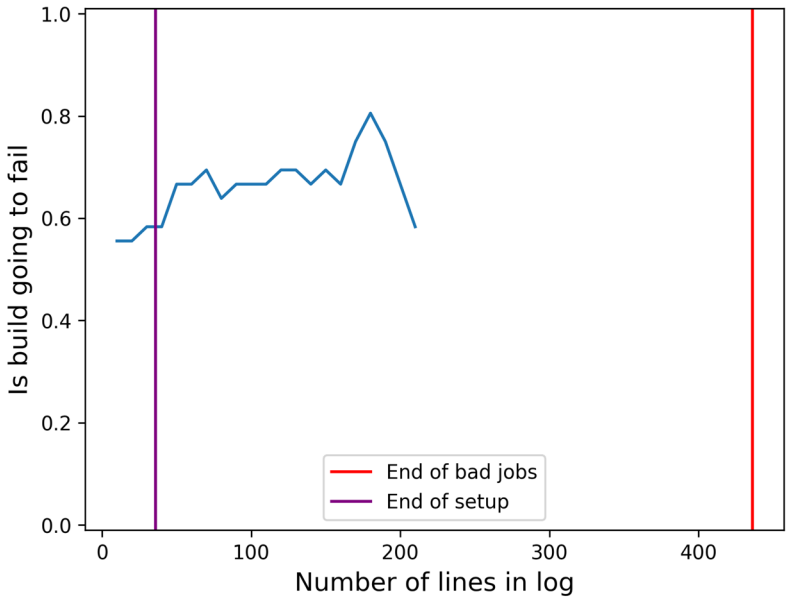


Figure A.2: Showing how likely the prediction system thinks the build is going to fail after how many lines of log is printed to the log.

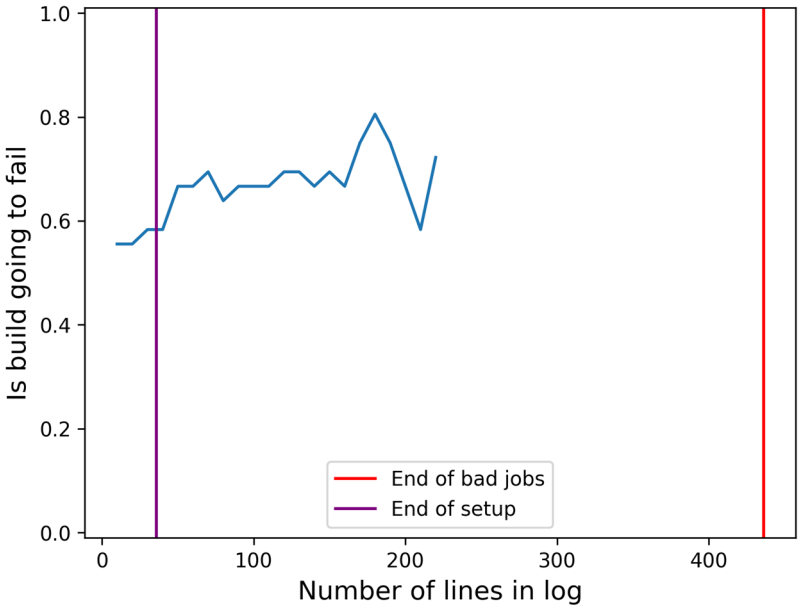


Figure A.3: Showing how likely the prediction system thinks the build is going to fail after how many lines of log is printed to the log.

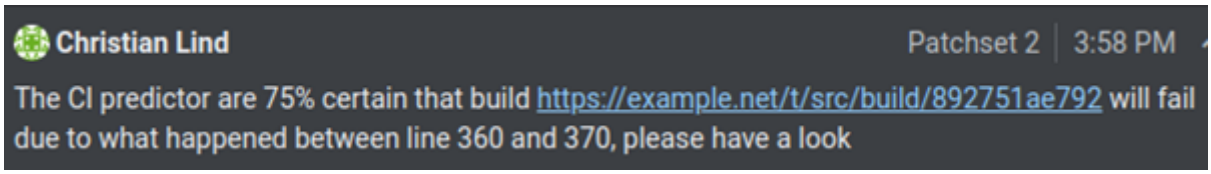


Figure A.4: Link to CI system inside Gerrit commit comment leading to marked text.

Figure A.5: Example of log text that are marked in red as the prediction system thinks this will cause the build to fail.

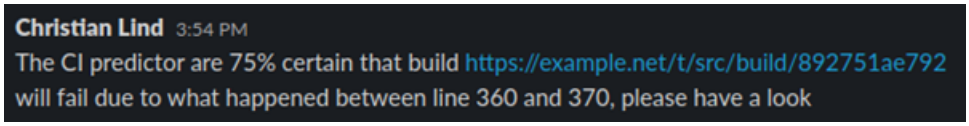


Figure A.6: Example message sent to the user from a botwith a link leading to an artifact in the CI system.

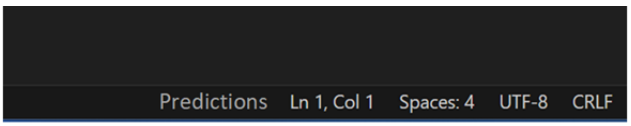


Figure A.7: Prediction button when no prediction is available.

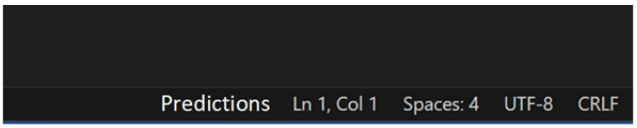


Figure A.8: Prediction button when a prediction is available.

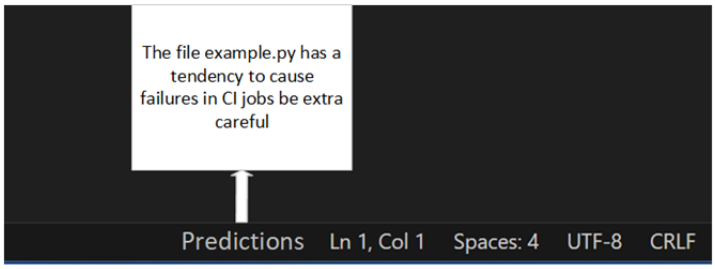


Figure A.9: Message shown to the user when clicking the predictions button.

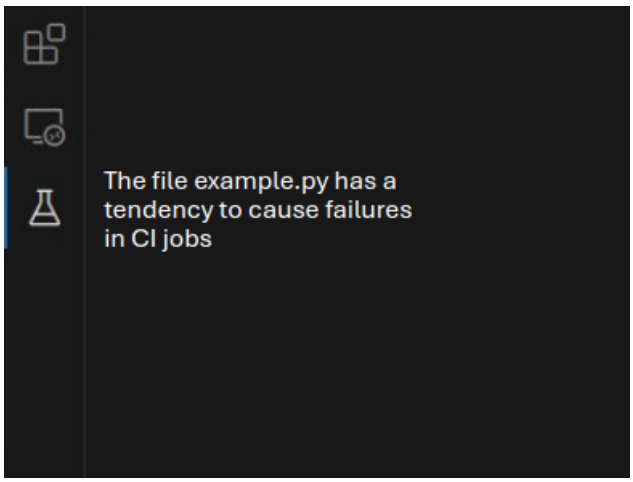
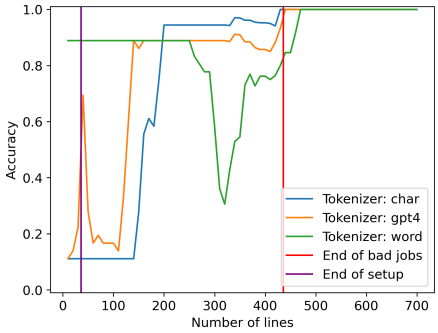


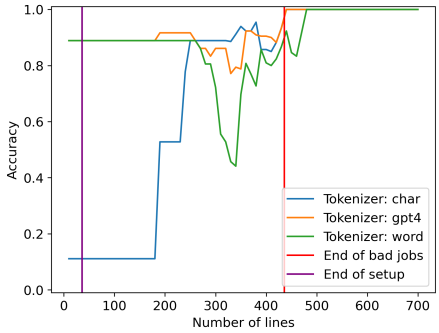
Figure A.10: Icon to notify the developer a prediction for the edited code is avaliable.

B

Appendix 2

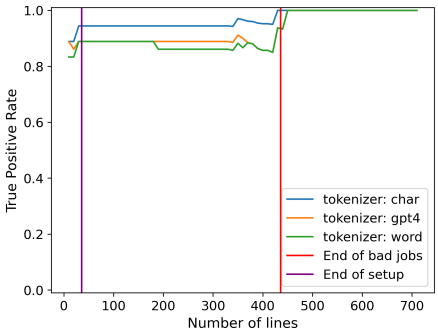


(a) Classifier: RF.

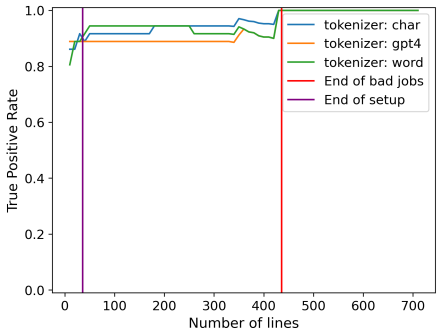


(b) Classifier: GB.

Figure B.1: Comparison of different tokenizers on Whole Log model with accuracy.

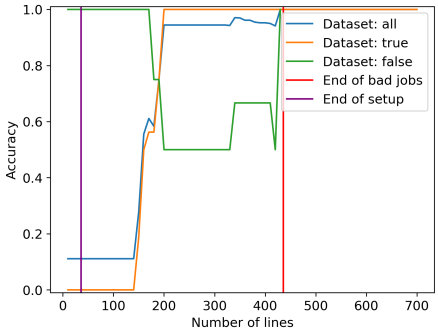


(a) Classifier: RF

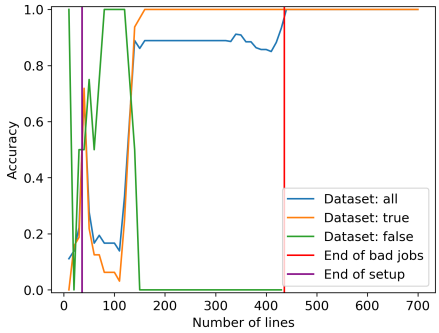


(b) Classifier: GB.

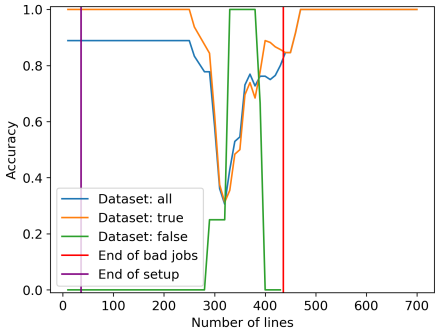
Figure B.2: Logtime model's accuracy using several types of tokenizers.



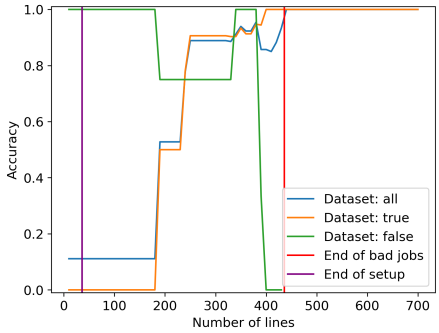
(a) Classifier: RF, Tokenizer: char.



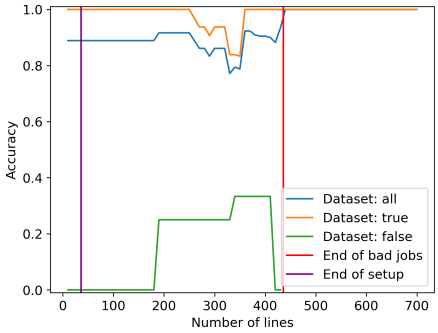
(b) Classifier: RF, Tokenizer: GPT4.



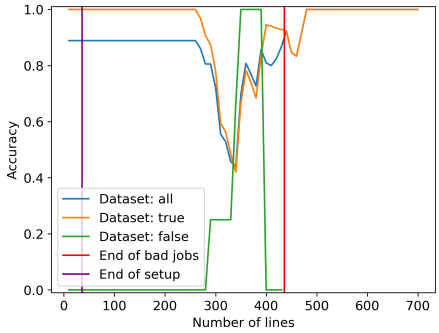
(c) Classifier: RF, Tokenizer: word.



(d) Classifier: GB, Tokenizer: char.

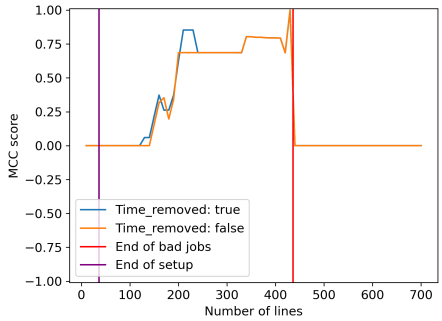


(e) Classifier: GB, Tokenizer: GPT4.

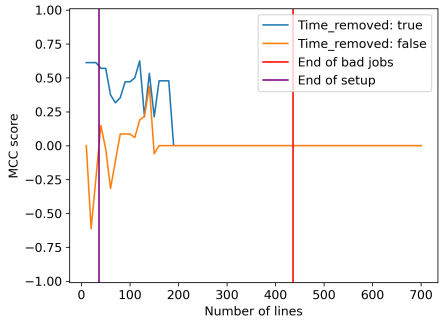


(f) Classifier: GB, Tokenizer: word.

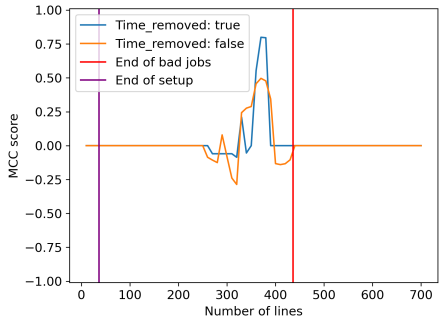
Figure B.3: Difference between predicting unsuccessful and successful jobs running on the Whole Log model with different tokenizers.



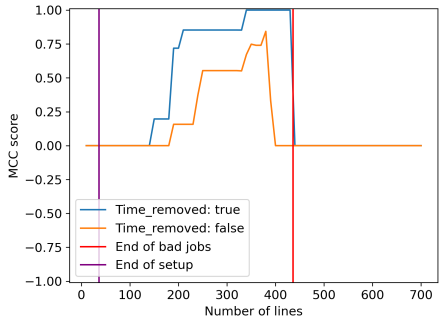
(a) Classifier: RF, Tokenizer: char.



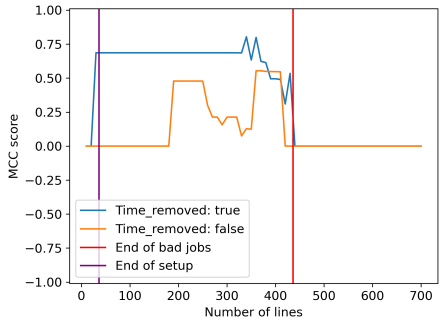
(b) Classifier: RF, Tokenizer: GPT4.



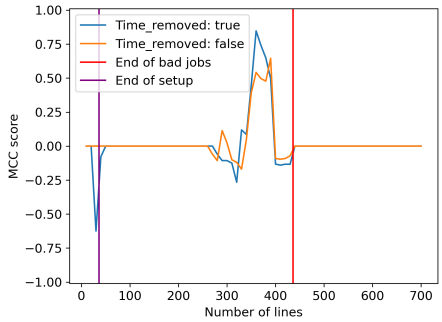
(c) Classifier: RF, Tokenizer: word.



(d) Classifier: GB, Tokenizer: char.

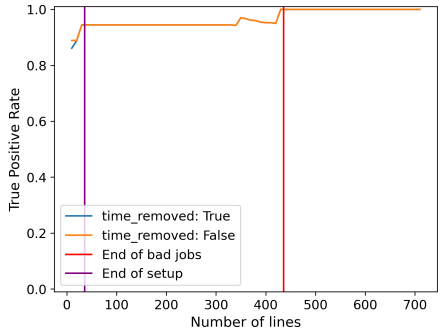


(e) Classifier: GB, Tokenizer: GPT4.

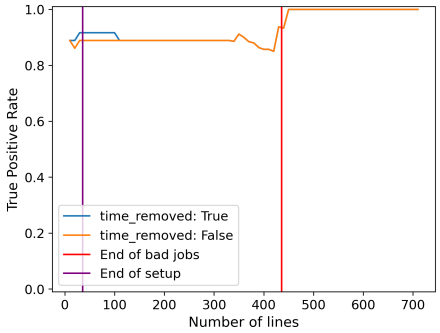


(f) Classifier: GB, Tokenizer: word.

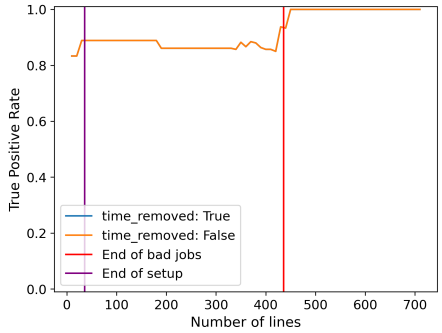
Figure B.4: Impact on MCC score when keeping and removing the timestamp from the logs running on the Whole Log model.



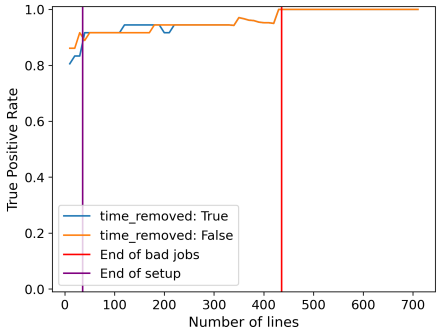
(a) Classifier: RF, Tokenizer: char.



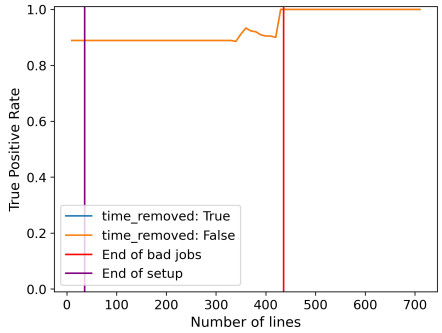
(b) Classifier: RF, Tokenizer: GPT4.



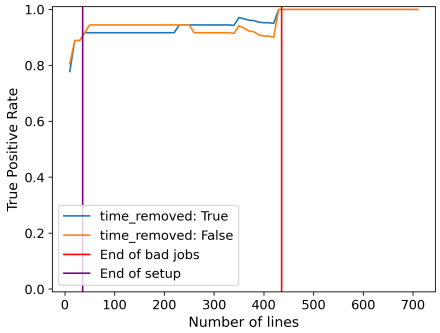
(c) Classifier: RF, Tokenizer: word.



(d) Classifier: GB, Tokenizer: char.

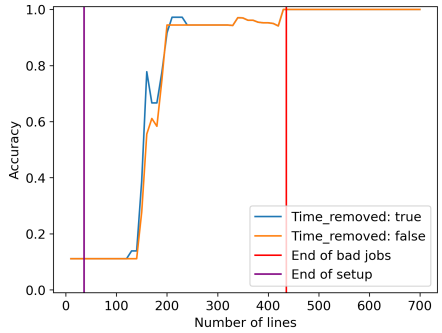


(e) Classifier: GB, Tokenizer: GPT4.

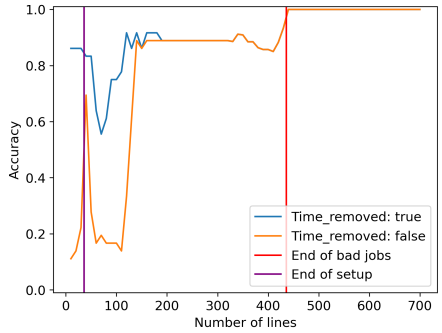


(f) Classifier: GB, Tokenizer: word.

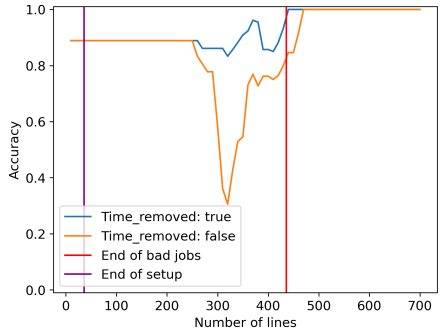
Figure B.5: Impact on accuracy when keeping and removing the timestamp from the logs running on the Logtime model.



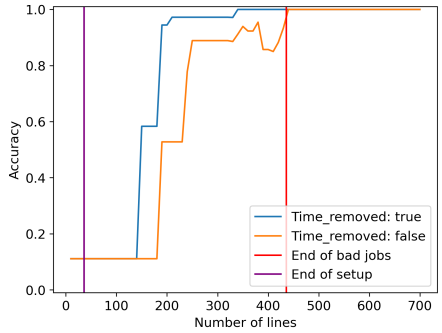
(a) Classifier: RF, Tokenizer: char.



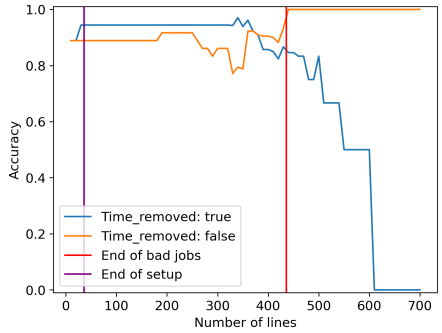
(b) Classifier: RF, Tokenizer: GPT4.



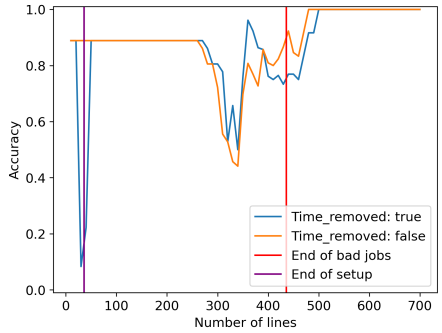
(c) Classifier: RF, Tokenizer: word.



(d) Classifier: GB, Tokenizer: char.



(e) Classifier: GB, Tokenizer: GPT4.



(f) Classifier: GB, Tokenizer: word.

Figure B.6: Impact on accuracy when keeping and removing the timestamp from the logs running on the Whole Log model.