MASTER'S THESIS 2022

Classification of Pull Requests using Transformers

Oscar Fridh, Szymon Stypa

ISSN 1650-2884
LU-CS-EX: 2022-33

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-33

Classification of Pull Requests using Transformers

Klassificering av kodändringar med Transformers

Oscar Fridh, Szymon Stypa

Classification of Pull Requests using Transformers

Oscar Fridh
os5614fr-s@student.lu.se

Szymon Stypa
sz6102st-s@student.lu.se

June 23, 2022

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Pierre Nugues, pierre.nugues@cs.lth.se
Oskar Handmark, oskar@backtick.se

Examiner: Jacek Malec, jacek.malec@cs.lth.se

Abstract

Pull-based development is a widespread paradigm in both open source and proprietary software projects. Since pull requests are a central piece of this workflow, their natural language descriptions are often key to ensuring an organized project progression. This thesis explores the usage of BERT based models for automatic pull request classification, and compares how models pre-trained on code and natural language react to different components of a pull request. We extended an existing dataset of 38,500 pull requests with additional features. We also collected and manually annotated a new test set of 500 pull requests. Using these datasets we fine-tuned and evaluated multiple transformer bases on different compositions of features and hyperparameters. We first show that the transformer models can reach higher F1-scores than the previous FastText classifier from DeepRelease when using the same input features. The results improved further when extending the inputs with additional data, allowing our best ensemble classifier to achieve a macro average F1-score of 0.63. Surprisingly, we find that the models pre-trained on code perform similarly or only slightly better than those trained on natural language when classifying code diffs.

Keywords: NLP, transformers, deep learning, pull requests, MLonCode, CodeBERT, CodeBERTa, BERTOverflow, DistilBERT, DeepRelease

Acknowledgements

A special thank you goes out to our supervisor Pierre Nugues for your continuous support and guidance throughout this project. We greatly appreciate our weekly meetings together and the time you have devoted to helping us.

We would also like to thank Backtick Technologies for allowing us to pursue this project and for everyone to take an interest in our work, giving us valuable feedback, while getting insight into a very inspiring workplace.

Finally, we would like to express our gratitude to the authors of DeepRelease who were kind enough to supply us with the dataset used for training their category discriminator along with their resulting fastText model.

Contents

Chapter 1
Introduction

1.1 Background and Context

Working as a software developer generally involves a heap of responsibilities, which do not entail development, but facilitate usage and further growth of a project. These might take the form of composing and maintaining a documentation, reviewing contributions, or writing release notes. More often than not, these responsibilities are seen as tedious tasks and constitute the least favored part of a developer's work. This results in a tendency for such tasks to get overlooked and creates a range of potential problems for project progression as a whole.

Backtick Technologies has been exploring ways to offload developer responsibilities by means of automation. The company is currently developing a product which enables generation of documentation pages from text content in the code base. In practice, the product takes the form of a web application which can synchronize with services like GitHub or Bitbucket and pull documents from repositories in order to host them through the service. This approach eliminates the need to build custom solutions which elegantly presents the information. Furthermore, since the documents are version controlled together with the code they describe, the assumption is that they will also be easier to maintain.

Naturally, the final level of sophistication for this kind of service would be to generate the documentation content automatically, without involving the developers at all. This is by no means an easy task, as code alone is rarely unambiguously tied to the contents of its documentation. Factors like design choices, mental frameworks, and development philosophy all have great impact on the final form of documentation. With this in mind, the more approachable variant of this problem is to work with an existing documentation structure and update or add entries based on changes committed to the code.

One essential milestone needed to achieve this functionality is the ability to automatically determine the nature of a change in the code base. Certain updates to a code base, for example new features, should almost always issue a documentation update which describes the new

functionality and how to use it. Other updates such as bug fixes, are less likely to require new content in the documentation, but might yield changes to existing entries.

This Master's thesis explores the problem of automatic classification of code changes, with the goal of facilitating the development of more sophisticated solutions for automated documentation maintenance. More precisely, we compare possible input features from pull requests and investigate several different pre-trained transformer bases and their performance on this downstream task.

1.2 Pull Requests

Pull-based development is a widespread paradigm in both private and open source projects. In this workflow, contributors create a feature branch (or fork) of a central master version when project content needs to be augmented. This enables individual tasks to be solved in isolated environments without impacting the functionality of the base state. Upon completion of such a task, the branch is submitted for review in the form of a *pull request* (PR).

The review essentially aims to determine whether the task has been solved safely and appropriately, and is often carried out by other project contributors. If approved, the branch is merged back to the central version, the PR is closed, and the feature branch is deleted. The main advantage of such a process is that consequently, contributors never submit code directly to the working state of a project. This minimizes the risk that a change breaks the repository and ensures that the master branch only contains finished and collectively approved work (Yu et al., 2016).

In order to easily review a PR, the PR usually gets assigned some metadata such as a title and body, describing the nature of its contribution. The PR can also be assigned a label from a set of categories. These labels are rarely standardized and may differ substantially between repositories. However, most projects do have overlapping or equivalent labels corresponding to the relatively typical categories of changes such as bug fixes, new features, or documentation changes.

1.3 Problem

In this thesis, we explored the use of state-of-the-art (SOTA) transformer models to automatically label changes in code. Since such changes have an intricate relation to their natural language descriptions (e.g labels or documentation), we also use pull requests and their corresponding natural language metadata as a proxy for modifications to code.

Why is it Important? Tools which automate tedious tasks are great for increasing developer productivity, as they free up the developer's attention to focus on more important tasks while relying on the tools. Our research aims to enable more automation in this regard and curtail developer responsibilities. Furthermore, it is common for developers to forget to label a pull request, and different open source projects use different labels. Developing an automated and standardized way of solving this burden also has the potential to improve project progression as a whole. Automatic labeling of pull requests could be used directly as a

feature in Docks, a tool for automating tasks tied to writing and maintaining documentation, developed at Backtick Technologies.

Scope. Our approach focuses on pull requests from open source repositories on GitHub as a proxy for changes to code. More specifically, we use the pull requests included in the dataset provided by DeepRelease (Jiang et al., 2022) together with a gold standard test set that we annotated for this thesis (see Sect. 2.3). The features that we explore in our experiments are: title, body, commit messages as well as code changes for each pull request. We examine several pre-trained transformer models based on the BERT and RoBERTa architectures with varying inputs and hyperparameters on the downstream task of pull request classification.

Goal. Our goal is to expand on the research of DeepRelease and hopefully contribute with even better results using BERT based transformer models. A central part will be exploring how transformers pre-trained on different types of data react to fine-tuning on specific input types like natural language and code.

DeepRelease chooses fastText over BERT for faster inference and deployment. In their case, the pull request classification is part of a larger system and needs to be fast. Our research has a relaxed time constraint and will evaluate classifiers based solely on accuracy and robustness – even though inference with these models is relatively quick.

1.4 Previous Work

There has been lots of research on the NLP classification task throughout the years, including a fair amount of research in the domain where code and natural language meet. This domain has recently started to see new models emerge that are based on the transformer architecture (Vaswani et al., 2017). This section describes previous work that we found particularly relevant for our thesis, listed in chronological order.

Pull Request Classification. Antoniol et al. (2008) demonstrated that it's possible to automatically classify an issue in an issue tracking system as either "bug" or "non-bug" based solely on the text of the issue. They used decision trees, naive Bayes classifier and logistic regression in their experiments, and were able to achieve precision between 0.64-0.98 and recall between 0.33-0.97.

Yu et al. (2018) proposed a supervised classification model combined with supervised topics model (LDA) and a naive Bayes classifier in order to automatically label (classify) pull requests on GitHub with an average precision of 0.6.

CodeSearchNet. Husain et al. (2019) created the CodeSearchNet Corpus, which is a large bimodal dataset of 2.1 million functions in six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby) along with their corresponding documentation, originating from a larger unimodal dataset of 6.1 million functions scraped from GitHub. On top of this, they presented the Code Search Challenge, which was a challenge on semantic code search. Their contributions have enabled lots of further interesting machine learning research on code and natural languages, particularly in the field of deep learning.

CodeBERT. Feng et al. (2020) introduced CodeBERT, a transformer model (limited to an encoder) that has been trained to learn representations from both programming languages (PL) and natural language (NL). It has been pre-trained on both the 2.1 million bimodal datapoints and the 6.4 million unimodal datapoints from the CodeSearchNet corpus in the tasks of masked language modeling (MLM) and replaced token detection (RTD). They showed empirically that CodeBERT outperformed other models in the downstream task of code search.

They also created a dataset for NL-PL probing [1], based on data from the CodeSearchNet, to investigate what type of knowledge CodeBERT learns. CodeBERT outperforms baselines on both NL and PL probing for all six languages. CodeBERT also achieved SOTA performance in the task of documentation generation. Furthermore they demonstrated in the task of generating NL summaries from C# that CodeBERT can generalize to unseen languages. Their research demonstrates that CodeBERT is able to learn implicit alignments between NL-PL as a general purpose representations that can be used for various downstream tasks that require NL-PL understanding.

CommitBERT. CommitBERT (Jung, 2021) is a BERT transformer model (encoder-decoder) for automatically generating commit messages based on code changes. They demonstrate that the large gap in contextual representation between code and natural language can be bridged using CodeBERT. They further demonstrate that it's better to pass only changed lines of code (additions and deletions) rather than the entire git diff into the model. They also contribute with a new dataset consisting of 345,000 code modifications from six programming languages (Python, PHP, Go, Java, JavaScript and Ruby) along with their corresponding commit messages, which they were able to retrieve from GitHub using the CodeSearchNet dataset.

Evaluating CodeBERT. Karmakar and Robbes (2021) evaluated CodeBERT, CodeBERTa, and GraphCodeBERT using probing to investigate if the encoders have actually managed to encode sufficient knowledge of code. This was evaluated by comparing them to a regular BERT model (trained on text). The difference in performance was slim, suggesting that future research on how to properly embed code knowledge into BERT encoder models is needed.

DeepRelease. DeepRelease (Jiang et al., 2022) is a newly developed system which automatically generates release notes based on pull requests. A central component in the system is a fastText model referred to as category discriminator. In essence, this is a pull request classifier which outputs relevant, standardized labels based solely on the title of a pull request.

Before development, the team conducted a study of release notes to identify the most common categories of change used in open source projects. Based on change frequency, they selected four category labels as a tagset for their classifier. To extend on their research and make our results comparable, our classifier will use the same four categories, namely: fix-bug, non-functional, new-feature, and documentation.

[1] A task similar to MLM, in which the goal is for the model to correctly predict/recover a masked token of interest from a pair of NL-PL tokens.

1.5 Contribution

Code is typically developed incrementally in cohesive units that can be categorized according to the nature of their contribution. Pull requests are commonly used in modern development workflows and enable individual tasks to be solved in isolated environments without impacting the functionality of the base state of the repository. They are typically opened on services like GitHub or Bitbucket, where they can be assigned metadata like descriptive texts and category labels. The vast amount of available open source repositories, along with their pull requests, make up a huge data source with the potential to enable lots of interesting research.

However, the pull request category labels are not standardized. Instead, they vary on a project-to-project basis and have to be manually assigned by developers. This task, although simple, is both tedious and prone to get overlooked.

In this thesis, we show that the BERT based transformer models are capable of automatically classifying pull requests into four predefined mutually exclusive labels. We achieved better results than the previous fastText model from DeepRelease (Jiang et al., 2022) on our own annotated dataset (see Sect. 2.3) using only pull request titles as input. The classification results improve further when combining more features such as code diffs and commit messages, allowing our best model to achieve a macro-average F1-score of 0.63. Furthermore, we found that transformers pre-trained on code performed similarly on the classification task compared to DistilBERT – regardless of the input being natural or programming language.

Our research demonstrates that transformer models are suitable for code related NLP tasks such as classifying pull requests. We anticipate our thesis to encourage more research in the field of automating tedious developer tasks using NLP techniques. For example, automatic classification of pull requests would most likely be a useful feature in more advanced challenges, such as finding discrepancies between code and documentation.

Distribution of Work. The work has been done in collaboration, and more or less all sections have been written by both authors.

Chapter 2

Datasets

2.1 DeepRelease

The DeepRelease dataset is derived from around 38,550 semi-automatically labeled PRs, but has been expanded by the authors to contain 10 times more entries through means of augmentation. Each datapoint in this set is comprised of a repository name, PR number, PR title and a category label. In its final form, it consists of around 385,500 datapoints with a label distribution which can be observed in Figure 2.1.

The titles in the dataset have been cleaned according to the rules described in DeepRelease, and the dataset has been split into training and validation subsets. The cleaned title entries are essentially distilled versions of the original PR titles, without any markdown or other formatting, and contain only natural language.

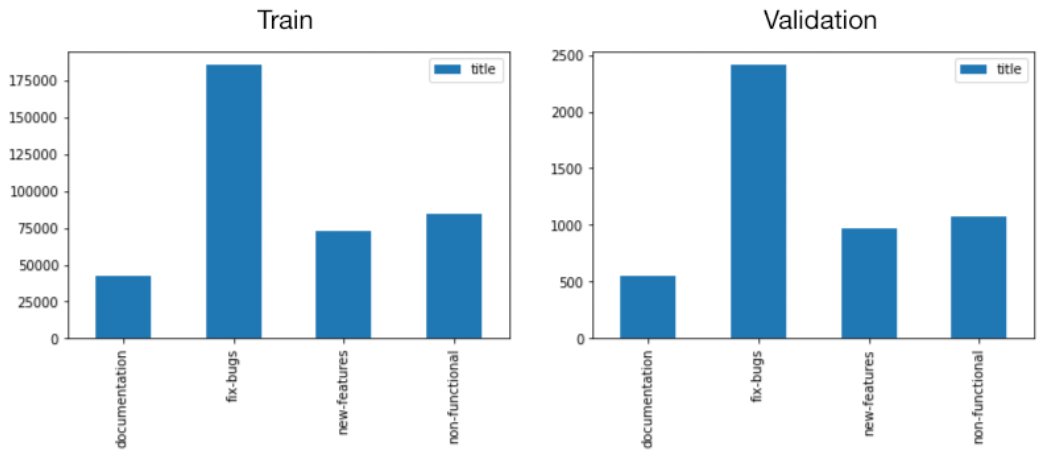


Figure 2.1: Distribution of categories in the data from DeepRelease

Although the datasets are not extremely unbalanced, we explored how their imbalance affects the performance in our experiments by training both using the datasets provided as is as well as balanced versions of the datasets. The balanced versions were created by undersampling the provided datasets so that each category only contains as many examples as the smallest category (documentation). These experiments are not mentioned in the evaluation part of this thesis, as we quickly realized that the resulting reduction in datapoints only impaired performance and decided to use the larger, original, distribution.

2.2 Scraped

As mentioned, the DeepRelease dataset is comprised of pure natural language titles. Because we were interested in exploring how additional metadata would affect the performance of our models, we needed an extended version of this set. Fortunately we were able to retrieve unique identifiers for each pull request by combining repository name and PR number in order to extend the datapoints. We used the GitHub API to scrape the original titles ourselves, along with the complete description body and the first 100 corresponding commit messages. This is the same approach that Jung (2021) used to create their dataset based on the CodeSearchNet corpus.

There are a couple of differences worth noting about our scraped data compared to the one provided by DeepRelease. The scraped entries in our dataset use the original text titles that can be found on GitHub. This was done intentionally so that the descriptions retain their formatting, which hypothetically could be a source of information for our models. In DeepRelease, the texts have been cleaned to remove anything that isn't natural language content. Their data has also been augmented to expand the dataset by a factor of 10, while our scraped dataset only has one datapoint per PR. Although more data is usually desired, we decided to not augment our scraped data. This decision was made due to time constraints as well as the tricky and erroneous nature of augmenting text data which isn't confined to natural language.

2.3 Our Gold Standard

The test set used for evaluation needs to be of highest quality in order to accurately assess the performance of a model. Since the DeepRelease dataset has been constructed using various rule-based automations, it is highly unlikely that it is entirely correct and was deemed inadequate for proper evaluation. Besides, we were interested in applying the DeepRelease fastText model on a dataset other than the one used in the original paper.

As we were unable to find any existing dataset suitable for our needs, we had to manually create our own by collecting and annotating data. We selected two familiar open source repositories and we labeled pull requests by manually looking at the title, body, commit messages, and code diffs. To somewhat speed up this process while not paying for existing solutions, we built a custom annotation tool for this using React.

The first repository selected for annotation was Cowait[1], an open source Python framework for creating containerized distributed applications developed by Backtick Technolo-

[1] `https://github.com/backtick-se/cowait`

gies. This was a natural choice for evaluation, as we ideally wanted our final model to perform well for Backtick employees. Secondly, we chose GRDB[2], which is a wrapper for sqlite written in Swift. Choosing Swift enabled us to evaluate if our models can generalize to unseen programming languages. We annotated around 250 pull requests from each repository, resulting in our gold standard dataset consisting of approximately 500 datapoints.

2.3.1 Inter-annotator agreement

An interesting note is that the DeepRelease fastText classifier performed very poorly on our gold standard dataset. Naturally, we had to find out if we used a different judgment for categorizing pull requests. In order to measure this, we first manually annotated 100 datapoints from the DeepRelease dataset. We then compared our annotations to the original labels and measured the inter-annotator agreement (IAA) using Cohen's kappa as metric. This resulted in a score of 0.51. Although it corresponds to a "moderate strength of agreement" according to the benchmarks by Landis and Koch (1977), we would have expected stronger agreements of 0.7 or higher.

2.4 Scraped Diffs

For our final experiment, we wanted to explore if the results would improve when using code changes as a model feature. We extended the scraped dataset with code diffs for each pull request scraped from GitHub. After discovering that some diffs are very large, we inspected their contents and came to the conclusion that we need to filter some of the entries. In order to avoid collecting metadata and focus on the actual code changes, we filter each diff to include only changes of files with certain extensions. The file extensions that we included are based on the languages used in DeepRelease and our gold standard set. Table 2.1 shows the list of them[3].

Language	File extensions
Python	.py
C / C++	.c .cpp .cc .h .o
C#	cs
Java	.java
JavaScript	.js .ts
PHP	.phtml .php*
Ruby	.rb
Go	.go
Swift	.swift
Web	.html .htm .xml .xlf .css .scss .sass .less
Other	.json .yaml .yml .md .rst .txt .sh .sql

Table 2.1: Languages and file extensions

[2] `https://github.com/groue/GRDB.swift`

[3] The * character acts as a wildcard and matches any symbol

In order to reduce the number of tokens for each datapoint, we also decided to distill the dataset by constraining the diff entries to only contain file location, additions, and deletions. While losing potentially valuable context data, this compromise would enable us to tokenize the diffs without truncating large portions of the changes. To our surprise, we later found out that Jung (2021) demonstrated that including only modifications without the surrounding context can actually improve the results.

Chapter 3
Theory

In this chapter, we cover theory that we consider crucial for understanding this project. We describe the fundamentals of neural networks and how to prepare input data using vectorization. We extend this with the transformer architecture along with some implementations, the classification task and relevant metrics. This leads up to transfer learning, which explains the relation between transformers and classification. Finally, we describe how multiple networks can be combined into one using ensemble classification and the fastText model architecture, which is used as a baseline from previous work relating to this thesis.

3.1 Neural Networks

Neural networks have proved to be successful in many tasks within machine learning and NLP, including the task of classification. They are a foundation for the transformer models that we explore in this thesis. Although the neural network model was originally inspired by neuroscience (Fitch, 1944), it is currently better understood as a mathematical model that exhibits several desirable properties, like the ability to perform distributed computations (Russell and Norvig, 2016).

A central part of the neural network is the neuron (also called node). A neuron takes input from other neurons, each with an associated weight. The weighted sum of inputs is optionally passed into an activation function (along with a bias) which essentially works like a threshold. When using activation functions, the neuron is fired and passes its output along only if the threshold is exceeded (Figure 3.1). If no activation is used, the output of the neuron is simply the weighted sum of its previous connections. This way a network of interconnected nodes can be created.

Depending on how the neurons are connected, different architectures can be constructed for different types of learning. In this thesis we focus on the simple feed-forward network (FFN), where neurons are connected in a directed acyclic graph (DAG). The neurons are organized in layers so that each neuron is connected to every neuron in the adjacent layers,

as illustrated in Figure 3.2 (Russell and Norvig, 2016).

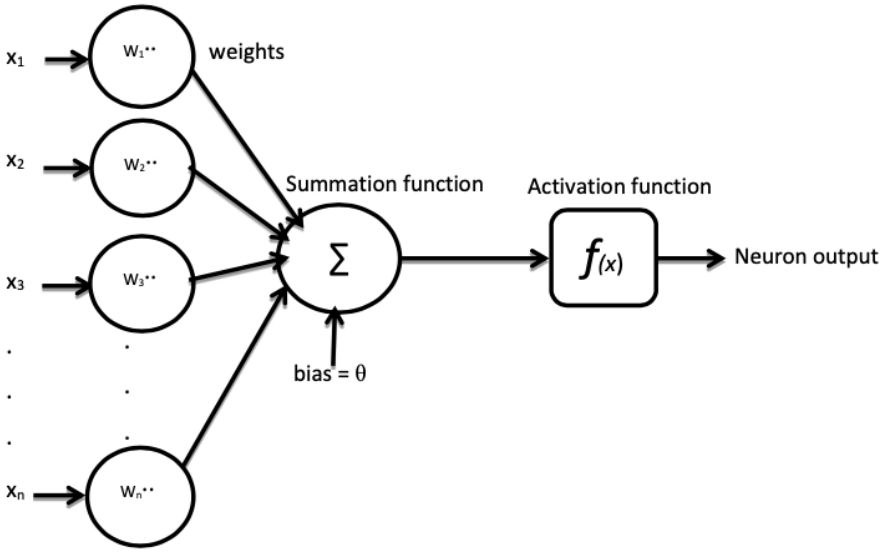


Figure 3.1: The structure of a single artificial neural network, from Yacim and Boshoff (2018).

The purpose of a neural network is (like any machine learning model in general) to find a function that accurately maps input data to output data. Simple linear combinations or complicated nonlinear functions can be modeled because of the activation functions. The model is trained by modifying the many (often millions of) parameters incrementally using gradient descent. In this procedure, the model's predictions are compared to the true values on random samples from the training data using a loss function. The loss function exists to express the performance of the model, and needs to be differentiable so that the gradient can be computed with respect to the model parameters. The parameters are then updated based on the gradient – a step which augments the mapping function towards its optimal value. This is commonly termed backpropagation and depends on the chain rule, which is necessary because of how the nodes are interconnected and influence each other.

3.2 Vectorization

Vectorization is the process of encoding all inputs into numeric tensors. This is necessary because neural networks, and machine learning models in general, only know how to deal with numerics (Chollet, 2017).

Tokenization. In order to effectively represent text data as numbers, we must first go through a process called *tokenization*. In this procedure, the text is divided into smaller parts called *tokens*. A token can be a word, a subword or a character depending on the strategy a model chooses to implement. By defining a fixed set of known tokens (a *vocabulary*), we can express a token numerically simply by referring to it by its position in this set.

Consequently, there is a trade-off between the length of a single token and the size of the vocabulary. If shorter tokens are used, the vocabulary needs less entries to cover all possi-

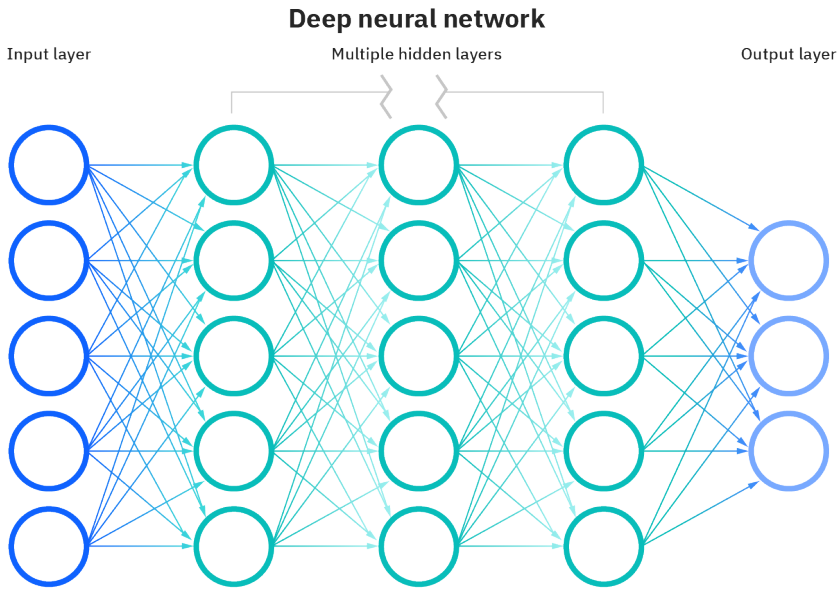


Figure 3.2: An illustration of a feed forward deep neural network by IBM (2020).

ble inputs – and vice versa. The precise tokenization strategy depends on the model, and is chosen to adequately encode the input types which a network will be trained on. The most common approach for the models used in our experiments is subword tokenization. However, models trained on code may implement different strategies to encode programming languages more effectively (Chollet, 2017).

Embeddings. By simply referencing tokens with their position in the vocabulary, we use what's commonly known as one-hot encoding. Unfortunately, this technique does not scale well and does not preserve the semantic relationship between tokens. A more current approach for numeric representation is using so-called embedding vectors – or simply embeddings. The idea here is to represent each token with a point in a large vector space. By training embedding models on large corpora, this type of representation is able to retain semantic relations.

A commonly used example to illustrate this property involves vector operations on the words "woman" and "king" in a 2D embedding space. In this scenario, shown in Figure 3.3, adding the embedding vectors for "woman" and "king" would result in the embedding for "queen." This has the additional consequence that semantic distance between words can easily be computed using cosine similarity (Chollet, 2017).

3.3 Transformers

The transformer architecture was first proposed in Vaswani et al. (2017) as a simplified alternative to recurrent neural networks (RNNs) and long short-term memory (LSTM). Just like these predecessors, the architecture is designed for processing sequential data. However,

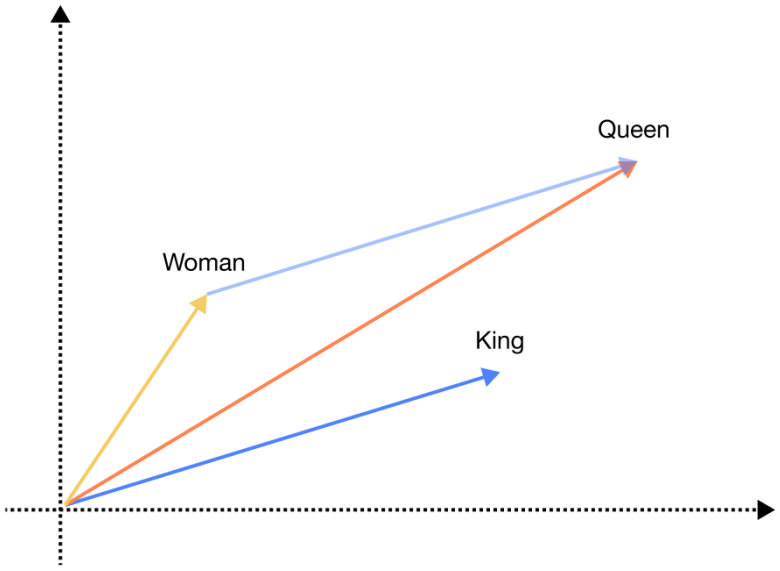


Figure 3.3: A conceptual illustration of how word embeddings can preserve semantic relationships.

transformers address several drawbacks of classical sequential models by relying on the attention mechanism and feed-forward layers instead of recurrence. As such, the need for feeding back states from previous inputs is eliminated. This results in easier parallelization, making the architecture cheaper and more effective to train. Furthermore, by processing whole sequences at a time, the network is able to learn arbitrarily long dependencies with ease.

Although the original paper describes transformers as a sequence-to-sequence network with an encoder-decoder structure (see Fig. 3.4), our focus will almost exclusively lie on its encoder portion. While there are plenty of interesting sequence-to-sequence applications for transformers, our problem is that of classifying sequences and does not require a decoder. Why this is the case will become apparent after Section 3.3.2.

The structure of the encoder is relatively simple, as it contains N (usually 6) identical blocks stacked on top of each other. Each of the blocks is comprised of self-attention and feed-forward layers, along with normalization steps which keep the values in finite ranges. Before the first encoder block, a positional encoding is added to the token embeddings to equip them with the positional information which is not apparent from embeddings alone.

3.3.1 Attention

The attention mechanism is the central building block of the transformer architecture, and enables learning of relations between elements in sequential inputs. It is, in its simplest form, a method of contextualizing sequences of vectors. While attention can be applied between two differing inputs, our previously targeted encoder part of a transformer applies attention on singular sequences to contextualize their elements – a process also referred to as *self-attention*. With this in mind, our descriptions will henceforth be limited to this specific type of attention.

Computing self-attention for an element is done by expressing a weighted combination of all other elements in the sequence. By attributing higher weights to more relevant elements, the mechanism is able to capture the importance of each element in relation to the whole sequence. The calculated attention can then be applied to the sequence vectors in order to "drag" each element towards the context of the sequence in the embedding space. In practice, this means that elements will be represented differently depending on the context they are in. To illustrate this intuitively, let us imagine we have two sentences containing the word "bank".

*I keep all my money in the **bank**.*
*I swam across the river to reach the other **bank**.*

It is evident that the word's meaning differs depending on which sentence we refer to. When putting the sentences through a trained encoder block, the intuition is that the vector representing the word "bank" will be shifted towards "money" and "keep" or "river" and "swam" respectively. That is if the attention layer has weights which are trained to contextualize sequences of English words (Vaswani et al., 2017).

3.3.2 Implementations

BERT

BERT is an acronym for Bidirectional Encoder Representations from Transformers. As the name suggests, it is a model architecture based on transformers which uses its encoder portion exclusively. It was first presented by Google Research in Devlin et al. (2018) along with SOTA results on various NLP tasks.

The paper presents an approach of pre-training transformer encoders using self-supervised tasks, which enables the model to robustly learn a general understanding of language. The pre-training is done using two tasks, Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). In MLM, randomly selected tokens are masked out and the model is tasked with predicting the missing token based on the left and the right context. The NSP task on the other hand, tasks the model with predicting the successor sentence for a given input. This fuses the left and the right context together, enabling the model to learn deep bidirectional representations.

Pre-training of this particular implementation is done on the English Wikipedia of 2,500M words as well as a book corpus of 800M words. This enables the encoder to capture a general sense of the structure of natural language. A BERT model pre-trained on this data can, in its final form, contextualize the input embeddings very effectively. Such a base network can then be extended and fine-tuned for a wide variety of downstream tasks, including classification (Devlin et al., 2018).

RoBERTa

RoBERTa is an acronym for: A Robustly Optimized BERT Pretraining Approach. Liu et al. (2019) demonstrated the importance of choosing the right hyperparameters when pre-training BERT with the following modifications:

- Train longer with bigger batches over more data
- Remove the NSP task
- Train on longer sequences
- Dynamically change masking pattern in MLM

Their research shows that the original BERT model was under-trained and that better results can be achieved using these modifications. The purpose of RoBERTa models is the same as BERT, it is simply an alternative approach which can give improved results (Liu et al., 2019).

3.4 Classification

One of the most common tasks in machine learning is classification. In order to perform classification, a model is trained to be able to assign a correct label based on a given set of input features. If the task is constrained to assigning a single class to each datapoint, it is referred to as *binary classification*. In this case, the model usually outputs a number representing its confidence in the assignment of this single class. If the model is to assign numerous classes, the task is instead referred to as *multiclass classification*. Here, the model usually outputs multiple values indicating its confidence in each of the classes.

Since we are working with four mutually exclusive categories, our downstream task is that of multiclass classification. In relation to transformers, this is accomplished by connecting the output layer of an encoder with a classification head (more about this in Section 3.5). The head is a linear layer that outputs the same number of nodes as the number of classes (in our case 4). After applying the softmax function, the output corresponds to the model's confidence that the input is of a certain class, interpreted as a probability (number between 0 and 1), for each class. Then the classification is just a matter of choosing the class with the highest confidence.

3.4.1 Evaluation

As classification is the primary focus of this thesis, we need some metrics to evaluate the performance of our classifiers. The most straightforward and intuitive way to express the models ability to assign correct labels is the accuracy metric, signifying the frequency of correct assignments:

$$\text{Accuracy} = \frac{\text{Correct Assignments}}{\text{Total Assignments}}$$

While this is an easily digested metric, it's not that descriptive when it comes to multiclass classification. Note that it does not express anything about the individual classes. Consider a model which simply outputs a single class regardless of the input. If all our datapoints belong to this fixed class, the accuracy for their classification would be a perfect 1 – despite the fact that the model is terrible at classifying and the data is skewed.

In order to define more sophisticated measures, we need to understand the concept of true/false positives and negatives. These are measures tied to each one of the specific classes, and are relatively self-explanatory once presented in this context. Examine the table below, where y and $\hat{y}$ correspond to a true and predicted class respectively:

	$y = 1$	$y = 0$
$\hat{y} = 1$	True Positives (TP)	False Positives (FP)
$\hat{y} = 0$	False Negatives (FN)	True Negatives (TN)

Having a total of four categories, we would assess the true and false negatives for a class by treating the 0 label as "any one of the three remaining classes." With these measures, we can define a couple of relevant metrics which help us concretize the performance of a multiclass classification model:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}},$$

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

While these are slightly less intuitive, they give us excellent ways to quantify the correctness of our classifications. Precision tells us how many of the items classified as a certain class actually belong to that class, whereas recall expresses how many of the items belonging to a certain class got classified correctly. By combining these two metrics in a harmonic mean, we arrive at arguably the most important metric in classification – the F1-score. This is the *de facto* standard metric for evaluating classifiers, giving us a robust metric which can be compared to previous results.

At this point, the attentive reader has noticed that the metrics covered above are calculated separately for each class, and do not reveal the overall performance on a multiclass task. While there are multiple strategies which combine them into singular numeric values, the most general and straightforward for our purposes is the macro average F1-score. As its name suggest, it is calculated by simply combining the F1-scores for each class into an unweighted average value.

3.5 Transfer Learning

Having covered transformers and classification, an additional piece of theory is needed to tie these concepts together – transfer learning. It is a concept which has gained huge popularity in the recent years, as it effectively reduces the amount of data needed to train an efficient model. The general idea is rooted in storing knowledge gained while learning one task and applying it on a different, related task. This can be done in many ways, depending on how much of the base knowledge that should be transferred to the target model.

One easy approach is to simply treat the output of an already trained network as an input to another during training. However, fully trained neural networks usually exhibit a behavior where the early layers of the network learn more general concepts, while the latter layers become more and more task-specific. This means that we can conceive countless variants of this approach. For example, only half of a pre-trained network could be used in order to utilize the more general layers of the network and discard the specifics. Moreover, we can let certain layers of the base network learn when training for the new task, while freezing others to conserve their original weights.

Taking computer vision networks as an example, the early layers will usually be more focused on detecting simple things like edges, while the later layers might detect shapes and finally complex patterns like objects or people. A great example of transfer learning is DeepLabCut presented in Mathis et al. (2018), which uses a neural network trained for human pose estimation as a base for animal pose estimation and achieves excellent results with minimal data.

In our case, knowledge will be transferred from transformer encoders trained on enormous corpora with self-supervised objectives (MLM, NSP) to a classification downstream task. Since the pre-training tasks for implementations like BERT and RoBERTa are focused on contextualizing inputs, the full models are usually utilized with an attached *head* – an extension of the network comprised of additional layers. The weights of these added layers must of course be trained on additional data. This process is referred to as *fine-tuning* a transformer, and will in our case be the training of a linear classification layer.

This raises the question: Should the previous pre-trained layers also be learning during fine-tuning and if so, which ones? The recommended strategy is to freeze the base of the network and gradually unfreeze more layers from the end of the network. The number of layers to unfreeze is a hyperparameter that needs to be found through experimentation. In general, the more training data we have, the more base layers can be included in fine-tuning (Chollet, 2017).

Sun et al. (2019) experimented with how to fine-tune BERT for text classfication. Layer-wise decreasing learning rate can overcome the catastrophic forgetting problem that can otherwise occur during transfer learning. HuggingFace makes this easy to implement using the provided learning rate scheduler.

3.6 Ensemble Classification

In order to use multiple features of different formats as input for classification, several classifiers (one for each feature type) can be combined and merged into a single multi-input classifier. This typically yields a better performance than any one of the models on their own, and can be applied to both binary and multiclass classifiers. The merging can be implemented in several ways – bootstrap aggregating, boosting and stacking being some popular methods. Aiming at simplicity, we decided to use stacking for our final experiment. It involves taking the outputs of trained models and training an additional layer on top of these outputs (see Figure 3.5). The network can alternatively be extended by additional layers on top of this, ending with a classifier. Chollet (2017)

3.7 FastText

Bojanowski et al. (2016) introduced fastText, developed by Facebook. It is not a deep neural network, but a simpler architecture that can be trained orders of magnitudes faster than deep neural networks while still achieving comparable performance. Joulin et al. (2016) trained a fastText classifier on more than one billion words in less than ten minutes on a CPU, and classified half a million sentences among 312,000 classes in less than a minute. This is currently far from possible using a deep neural network.

The fastText model generates word embeddings using n-grams. A word is represented as the sum of its n-grams, where each n-gram associated with a vector. By using vectors at the n-grams level instead of words, an internal structure of building blocks for words is preserved. This leads to better representations of rare unknown words where part of the n-grams building up the word is known. The benefit is stronger for certain languages that exhibits certain grammatical features like declensions and compound words, like German (Bojanowski et al., 2016).

Despite the promising research on fastText, we will only use it as a baseline in this thesis. Several BERT based transformer models will be trained and compared against the fastText model developed by DeepRelease (Jiang et al., 2022) in the task of classifying pull requests. A notable difference between fastText and BERT is that fastText embeddings are not contextualized.

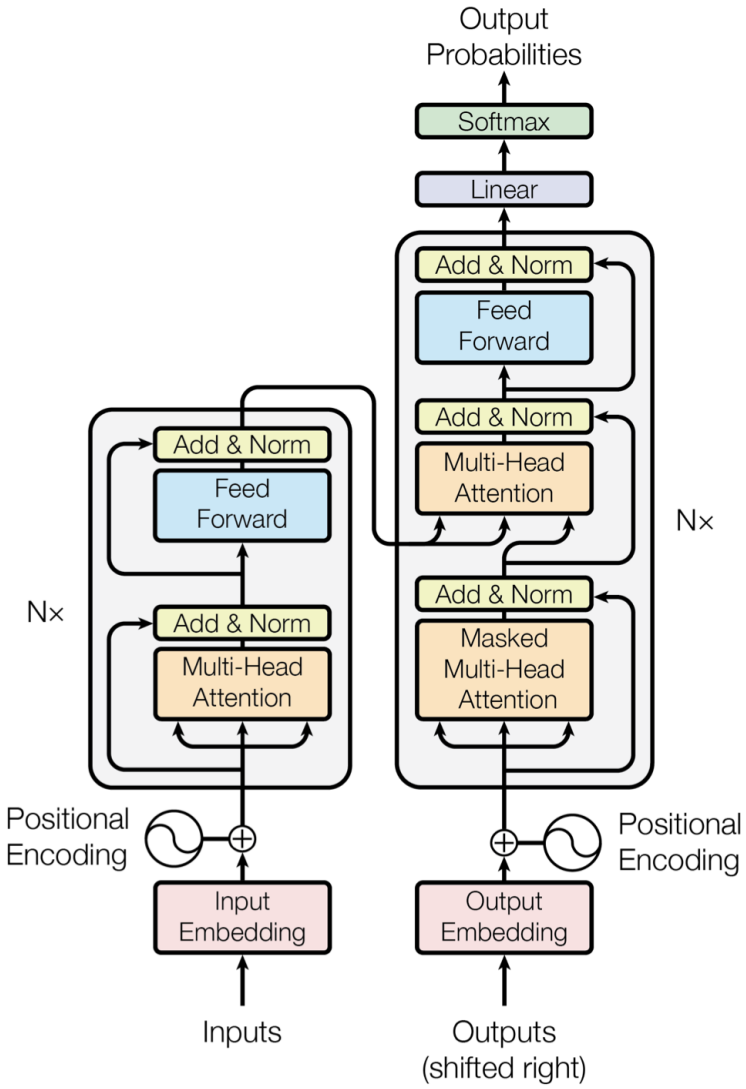


Figure 3.4: The transformer architecture from Vaswani et al. (2017). The left part depicts the encoder and the right part depicts the decoder. This thesis only focuses on the encoder. Instead of passing output from the encoder into a decoder we will pass it into a classification head.

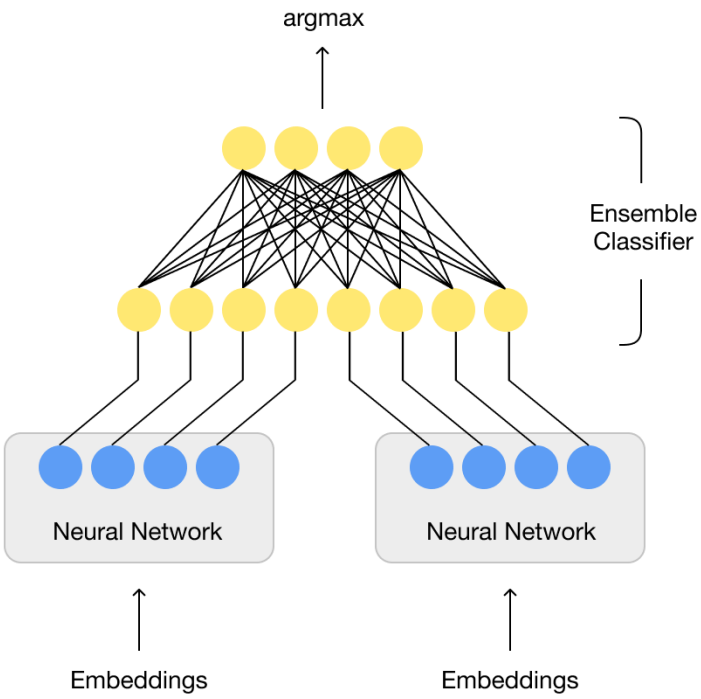


Figure 3.5: An illustration of the simplest possible combination of multiple classifiers. Two networks outputting four values are merged by stacking their last layers into a dense layer.

Chapter 4
Approach

4.1 Method

The methodology used to reach our results was based on IBM's Cross-Industry Standard Process for Data Mining (CRISP-DM). It is a proven data mining life cycle model consisting of six phases which ensure steady progression of any data mining project. The standard is flexible while offering effective guidelines for typical tasks in each phase. Moreover, it is not necessarily sequential which means that projects may iterate between the phases freely (IBM, 2021).

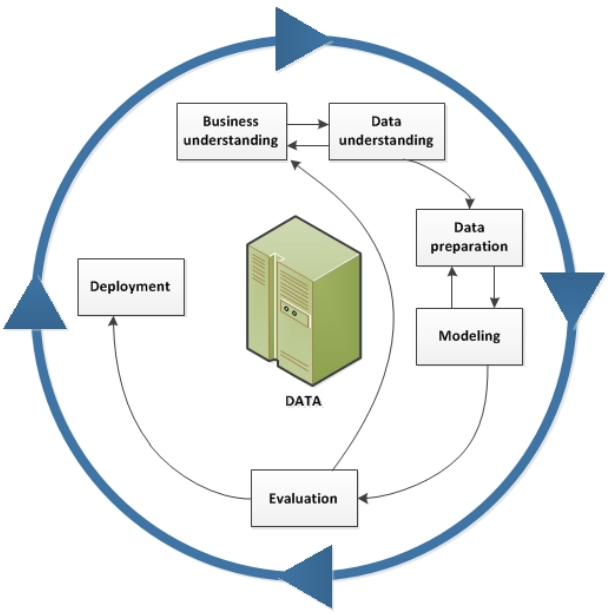


Figure 4.1: The CRISP-DM life cycle as depicted in IBM's documentation (IBM, 2021).

Business Understanding. The first phase focuses on exploring the business expectations of the project. By involving key partners, the goal is to establish what type of business result the investors hope to obtain. It is crucial that all participating for-profit parts and others are on the same page when it comes to what the project might generate (IBM, 2021).

Data Understanding. In order to understand existing and potential data sources, this phase focuses on exploring the attributes and nature of the data involved. At this stage, it is important to answer questions such as: Are there existing data sources? Which attributes of the data seem promising and which irrelevant? Is there enough data? Is the data of sufficient quality? Does additional data need to be collected?

Data Preparation. This phase is often the most burdensome and time-consuming aspect of data mining projects. As raw data can be obtained from virtually any source, it follows that it might need to be processed extensively to fit its target application. The processing is comprised of different augmentations to the data depending on the project. Generally, it will involve selecting (aggregating, splitting), cleaning, formatting and even composition of new attributes until the data is fit for model training (IBM, 2021).

Modeling. When data is prepared, it is time to put it to good use. At the modelling stage, different models are first trained on the data with default parameters in order to determine a good fit. This way, a general understanding of what architectures work can be obtained. The best models are then chosen for hyperparameter optimization until the models are effective and satisfactory results are obtained (IBM, 2021).

Evaluation. Evaluation looks different depending on the task a model is trained to solve. Generally speaking, several metrics should be obtained in order to properly evaluate the performance of models. In this phase, these metrics are computed and tied back to the goals obtained during business understanding. The main goal here is to establish whether the model does what it's supposed to do. If not, it is a perfect stage to revert back to previous phases in order to make sure that it does (IBM, 2021).

Deployment. In this final stage, the superior models are either formally integrated, or the insights gained from them are put to use. That could mean deploying the models for production inference or simply using the results to make improvements withing the organization (IBM, 2021).

4.2 Implementation

The language of choice for the entire implementation was Python because of its extensive supply of useful machine learning libraries. Early on, we wrote quite robust command line interface tools (CLI) for communication with GitHub through its REST API, web endpoints, and the git protocol. We utilized them to collect the gold standard dataset and extend the DeepRelease datapoints. In order to effortlessly annotate our collected data without paying for existing solutions, we also wrote a simple annotator tool with JavaScript and the React framework.

Tools. The HuggingFace model hub along with their Transformers library (Wolf et al., 2020) provided us with all of our pre-trained transformer bases. We built a fine-tuning procedure using a mix of components from Transformers and raw PyTorch. We made the decision not to use Tensorflow or the HuggingFace trainer API for these tasks due to model compatibility, flexibility, and complexity of GPU setup.

We used Matplotlib for plotting and Scikit-learn for evaluation metrics such as confusion matrices and classification reports. We wrote all our programs using Jupyter notebooks, with the exception of data gathering and certain formatting measures.

Hardware. Initially, our notebooks ran on Google Colab as it offers free Python environments with access to relatively powerful GPUs – without requiring any setup. After training a couple of models, we realized that the platform did not meet our needs due to the computing and time restrictions imposed by Google. In order to address this, we configured a personal gaming PC with an RTX 2080 Super GPU with CUDA as a dedicated remote machine for our training procedures. As it turns out, this decreased the training time by a factor of approximately 5 compared to Colab's Tesla K80 GPUs. It also enabled us to train without time or processing restrictions, making it painless to run large training jobs during convenient times without the need to maintain a network connection.

Preparation. In order to maximize the training speed, we prepared for training by:

1. Tokenizing each datapoint in the set and computing its length.

2. Computing the distribution of all tokenized lengths and the last ventile[1] breakpoint value in order to determine the token length which covers 95% of the set.

3. Applying a threshold with this value as the max length for the tokenizer with a minimum value of 32.

Since all our transformer bases have an input size of 512, this strategy gives significant speed improvements if working with short sequences. By limiting the length of input sequences to the model, less nodes are activated in the network and thereby the number of nodes involved in backpropagation is reduced. When training the later models with aggregated inputs and diffs, however, the ventile value is slightly above 512 and some inputs are instead truncated to fit the model.

Training. We start by shuffling and splitting the dataset into training and validation subsets with a ratio of 80/20. After tokenizing the entries, the selected transformer base is initialized with a classification head comprised of a linear layer converging the output of the transformer into 4 nodes. The model is solidified by freezing all base layers up to the linear classification layer, and an AdamW learning rate optimizer is instantiated with an initial learning rate of 10^{-3} (default). We then train for 10 epochs while saving the loss, accuracy and macro F1 metrics for both the training and validation sets. Additionally, we save the model state with lowest validation loss so far. The process repeats with the next transformer base using the same parameters.

[1] Quantile of order 20

When all transformer bases have been trained on the data with this base configuration, we continue into an intermediate evaluation step. By comparing the learning metrics and applying the lowest-loss model on the test set, we establish which transformer bases are performing best and discard those that perform poorly. Next, the training procedure for the most promising bases are repeated three times. Each time, another rearmost layer of the base encoder is unfreezed for backpropagation. An illustration of this process can be seen in Figure 4.2. Lastly, we compare each version's learning progress and try adjusting the initial learning rates for the champions as preparation for a final round of training with the most promising set of parameters.

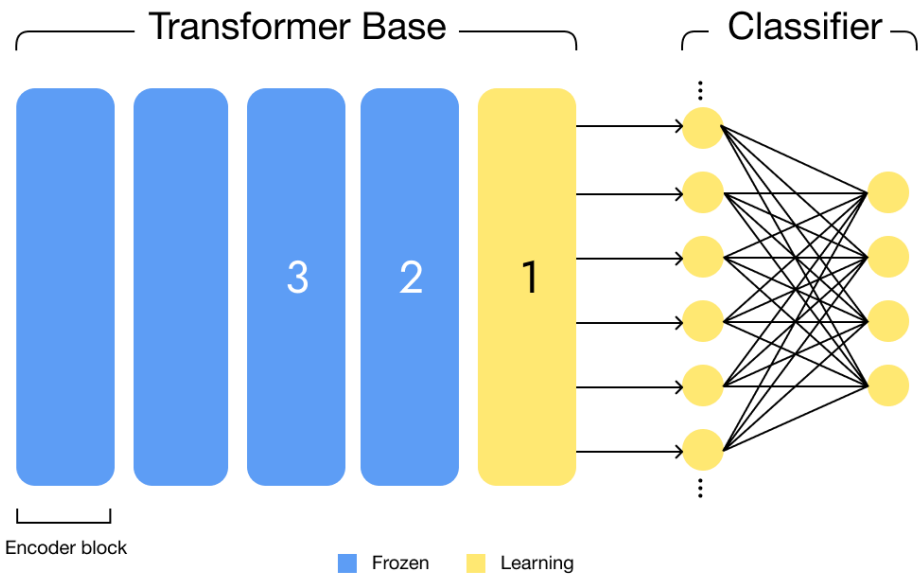


Figure 4.2: An illustration of the training steps when unfreezing layers of the base encoder. The state showed in the figure corresponds to the first training after evaluating the base configurations. Next round, block 2 will also be unfreezed for backpropagation.

Chapter 5
Models

All of the following models are available on the HuggingFace model hub. They have been chosen based on their type of training data and popularity. Their pre-training data ranges from natural language to code, with some being trained on natural language in a code context and others on a mix of both natural language and code.

5.1 DistilBERT

DistilBERT is a pre-trained encoder model trained on natural language in the same fashion as BERT, but with much fewer parameters. The authors were able to show in Sanh et al. (2019) that the BERT model complexity can be reduced by 40%, while retaining 97% of its language understanding capabilities and being 60% faster. This was accomplished using knowledge distillation, which is a compression technique for ML models. The motivation for their research was that smaller models are cheaper and faster to train with reduced environmental footprint. Figure 5.1 shows a graph of complexity vs release date for some of the most popular transformer bases from the last five years. It is evident that the complexity to performance ratio for this model is groundbreaking, even today.

5.2 CodeBERT

CodeBERT is a transformer that has been pre-trained on both programming and natural language, developed by a Microsoft research team in China. The model architecture is exactly the same as RoBERTa and has a total of 125M parameters. Pre-training was done on the CodeSearchNet corpus (Husain et al., 2019) which consists of 2.1 bimodal datapoints of NL-PL pairs as well as 6.4M unimodal PL datapoints, all across six programming languages (Python, Java, JavaScript, PHP, Ruby, Go). The pre-training is done on both MLM and NSP. CodeBERT achieves state-of-the-art performance on downstream tasks including natural language

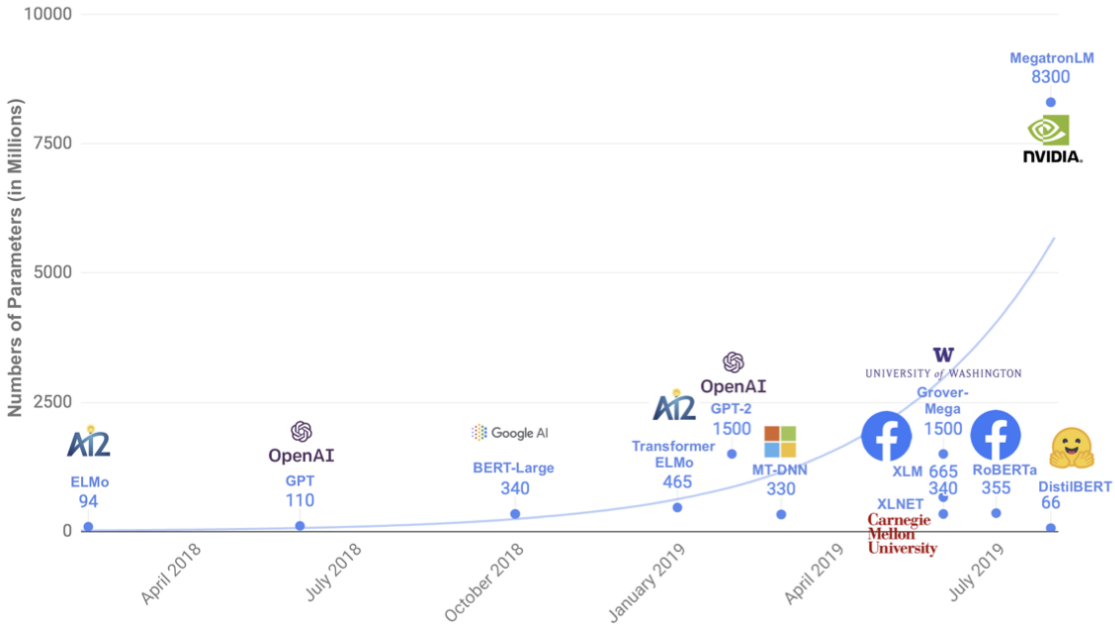


Figure 5.1: Number of parameters of various models, over time, from Sanh et al. (2019)

code search and code-to-documentation generation. The model also seems to generalize to other languages in their experiment on unseen data in C# Feng et al. (2020).

5.3 BERTOverflow

BERTOverflow is an encoder that has been pre-trained on a corpus of 152 million sentences from StackOverflow's 10 year archive (2008-2018). They evaluated several models on a manually annotated test set (of sentences from StackOverflow and GitHub) and showed that the BERTOverflow embeddings improved the results on the downstream task.

The StackOverflow NER corpus, used as test set, has been manually annotated by 4 students. 20 types of fine grained entities have been labeled on questions and answers from randomly selected questions as well as random sentences from GitHub readme files. The inter-annotator agreement Cohen's Kappa score before adjudication was 0.62.

Tabassum et al. (2020)

5.4 CodeBERTa

CodeBERTa is a RoBERTa-like model trained on the CodeSearchNet corpus. The tokenized corpus can be encoded more efficiently due to the structured and repetitive nature of code in contrast to natural language. Sadly, there is little to no information available as to the details of this model[1].

[1] `https://huggingface.co/huggingface/CodeBERTa-small-v1`

Chapter 6
Evaluation

We evaluate the DeepRelease classifier on our Gold Standard dataset as a first experiment (Sect. 6.1). We also train and evaluate our BERT classification models in a similar setting using only titles (Sect. 6.2). We then try to improve the results of our classification models by experimenting with different features. In one experiment, we use additional text as input (Sect. 6.3) and in another experiment we use only code-diffs as input (Sect. 6.4). Finally, we combine theses features in an ensemble classifier (Sect. 6.5).

6.1 FastText Classifier

We evaluated the fastText classifier provided by DeepRelease on our Gold Standard dataset as a baseline. This resulted in a macro F1-score of only 0.33, and the full classification report can be seen in Table 6.1. It is apparent that the model is very eager to assign the fix-bugs category, resulting in a low precision for the label. The new-features and non-functional categories are particularly likely to get misclassified as bug fixes, resulting in poor recall for both categories.

6.2 Titles

We then trained our classification models on the scraped dataset that only contains formatted titles (Section 2.2). A comparison of all the transformer bases with the base configuration can be seen in Figure 6.1. We observe a quite large difference in performance between the models, with DistilBERT and CodeBERTa outperforming the others by a large margin.

Optimization. Considering that DistilBERT and CodeBERTa performed well with the base configuration, we chose these two models for the second round of training. Following

	Precision	Recall	F1-Score	Support
fix-bugs	0.17	0.66	0.27	76
new-features	0.42	0.26	0.32	149
documentation	0.58	0.37	0.45	52
non-functional	0.46	0.19	0.27	228
accuracy			0.30	505
macro avg	0.41	0.37	0.33	505
weighted avg	0.42	0.30	0.30	505

		Predicted			
		fix-bugs	new-features	documentation	non-functional
True	fix-bugs	0.67	0.15	0.03	0.15
	new-features	0.58	0.26	0.03	0.13
	documentation	0.21	0.08	0.37	0.35
	non-functional	0.62	0.16	0.03	0.19

Table 6.1: Test set results for the baseline fastText classifier.

the procedure described in Section 4.2, we completed three rounds of training and unfreezing encoder layers for each model.

The configurations showing most promise were DistilBERT with two unfreezed layers and CodeBERTa with one. Since the training metrics seem stable, the learning rate was not adjusted. A comparison of the best configuration for both bases can be seen in Figure 6.2, where we observe that DistilBERT yields slightly better results on the validation set. This is also the case for the test set, where CodeBERTa achieves a macro F1-score of 0.47 and DistilBERT of 0.52. The complete test set classification results of the best performing model can be seen in Table 6.2.

6.3 Extended Text Data

We then used our scraped dataset with body and commit messages as features for each pull request in addition to title. We concatenated all inputs into a single string that is tokenized and passed into the models. The results for the base configurations can be seen in Figure 6.3, where we see DistilBERT performing quite well again. CodeBERTa, on the other hand, seems very unstable in its learning progress.

Optimization. After running the training procedure with additional unfrozen layers, we saw that both CodeBERTa and DistilBERT gained performance with one unfreezed layer. Also, when reducing the learning rate by half, we found that the CodeBERTa model gained some stability and that the DistilBERT results improved further. A comparison of the best models with one unfreezed layer and adjusted learning rates can be seen in Figure 6.4.

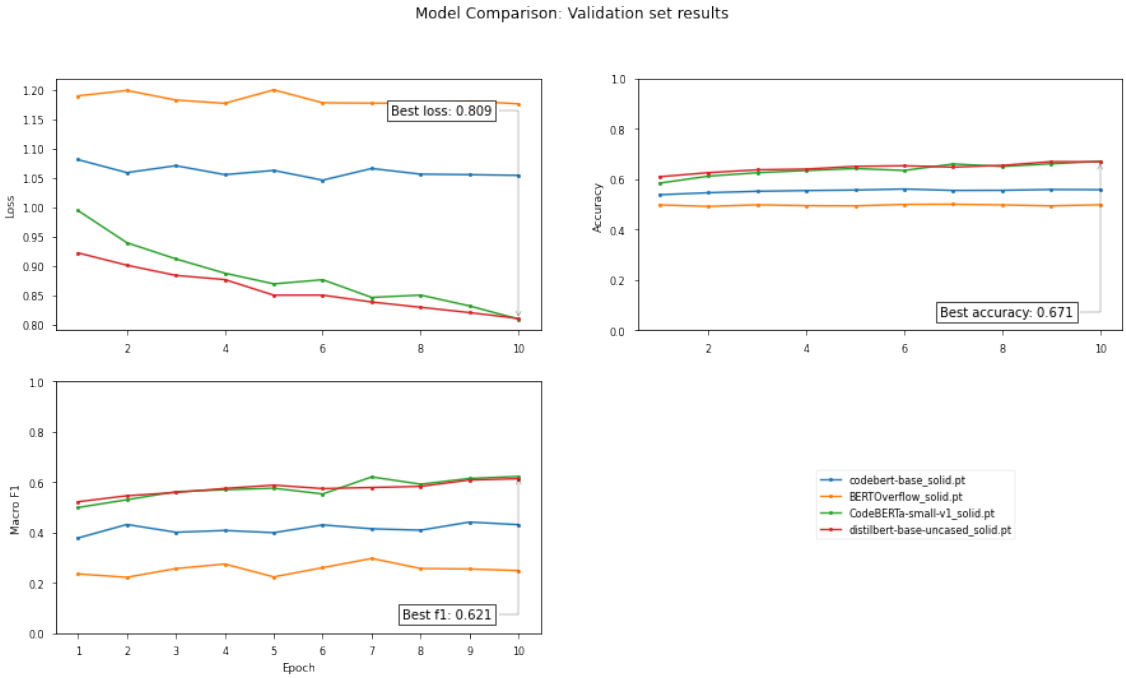


Figure 6.1: Comparison of each transformer with the base configuration - trained on titles

6.4 Code Diffs

In another experiment, we used our scraped dataset containing code diffs for each pull request (2.4). At first, we experimented with per-file prediction where each file entry in the diff was input and the parent PR category was the label. This approach seemed better, as it yields less truncated inputs. However, after comparing with models trained on entire PR diffs, it was clear that this strategy gave significantly worse results.

The comparison of each transformer base for per-PR prediction can be seen in Figure 6.5. Here, we actually see that CodeBERTa pre-trained on code yields slightly better results, reaching a macro F1 of 0.59 – but the CodeBERT model still lags behind DistilBERT.

Optimization. Surprisingly, we could not improve the results from the base config training of these models. A fully frozen CodeBERTa and DistilBERT with default learning rates were the best candidates, but ultimately CodeBERTa showed slightly better metrics (Figure 6.5). As can be seen in Table 6.4, the overall results for the best model were not very impressive. However, the accuracy is still better than pure guessing so the model should add some performance to the extended data classifiers when combined in an ensemble.

6.5 Combined Classifiers

The training of an ensemble classifier is preferably done on data previously unseen by the individual classifiers. Due to scarcity of such data, we tried training the ensemble classifier on both the training and validation (unseen) data. Interestingly, it showed better performance

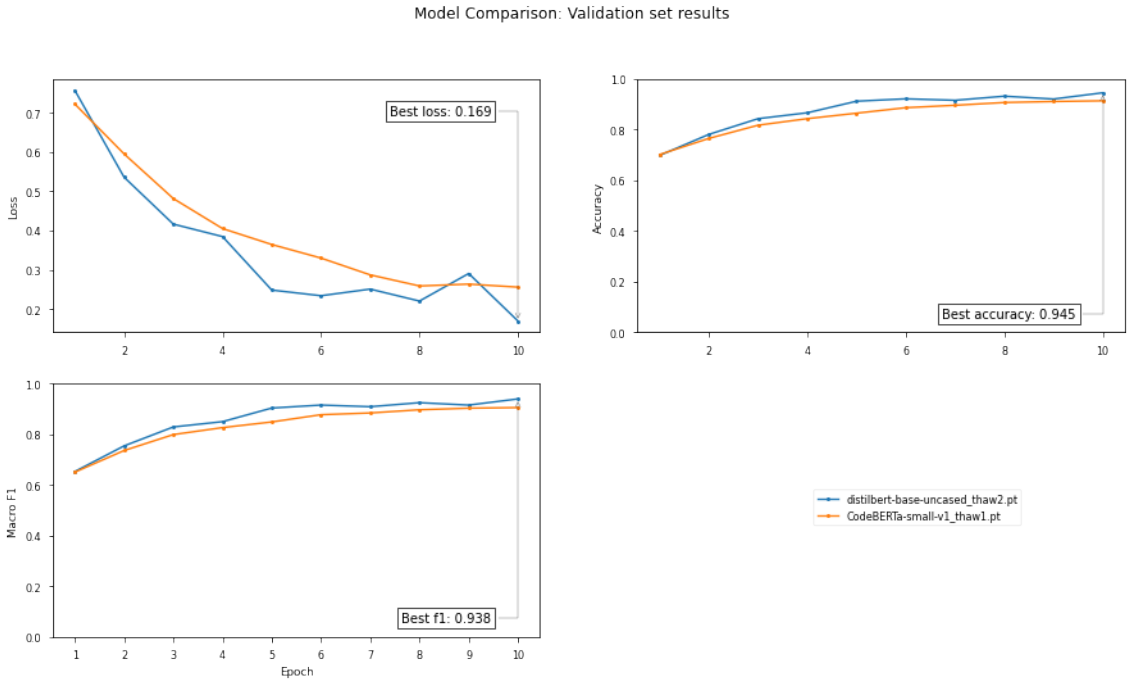


Figure 6.2: Comparison of the best DistilBERT and CodeBERTa versions - trained on titles

for the former. This might be due to the fact that the validation set was considerably smaller and was not enough to learn how to properly combine the network outputs.

Table 6.5 shows the metrics for the final ensemble classifier combining the DistilBERT model for titles, body and commits with CodeBERTa for code diffs. We eventually reached a macro F1 of 0.63 with our best model.

	Precision	Recall	F1-Score	Support
fix-bugs	0.33	0.87	0.48	76
new-features	0.50	0.36	0.42	149
documentation	0.67	0.65	0.66	52
non-functional	0.66	0.43	0.52	228
accuracy			0.50	505
macro avg	0.54	0.58	0.52	505
weighted avg	0.57	0.50	0.50	505

		Predicted			
		fix-bugs	new-features	documentation	non-functional
True	fix-bugs	0.87	0.04	0.03	0.07
	new-features	0.37	0.36	0.03	0.24
	documentation	0.15	0.02	0.65	0.17
	non-functional	0.32	0.21	0.04	0.43

Table 6.2: Test set metrics for DistilBERT with two unfreezed layers – trained on titles.

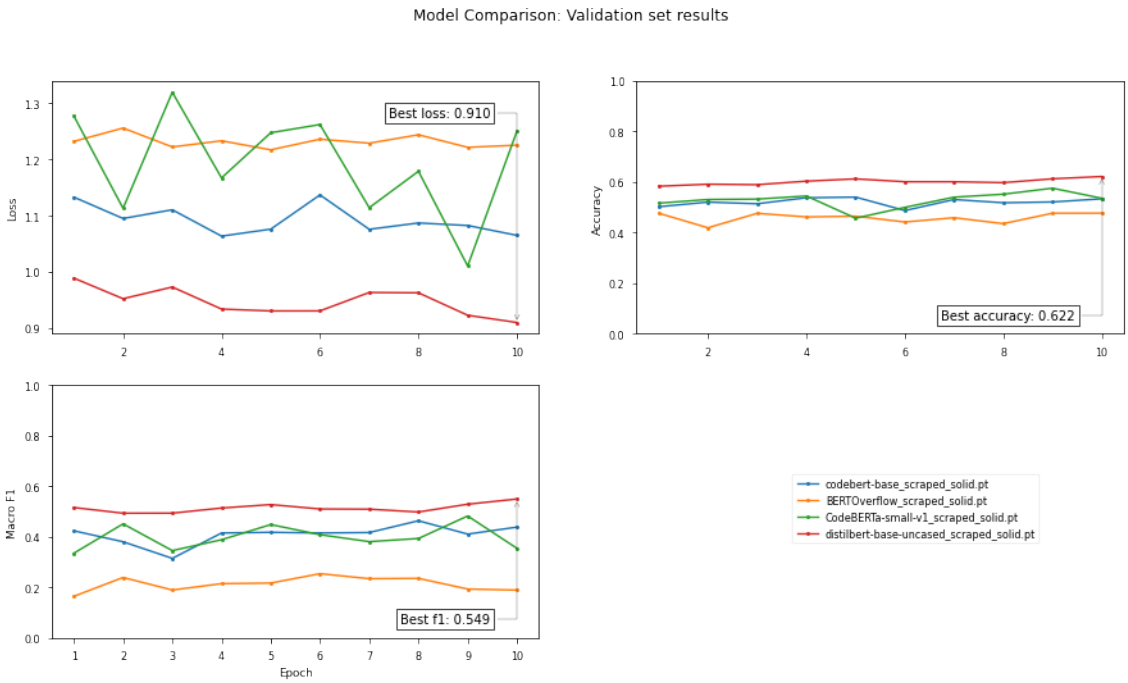


Figure 6.3: Comparison of each transformer with the base configuration - trained on extended text data

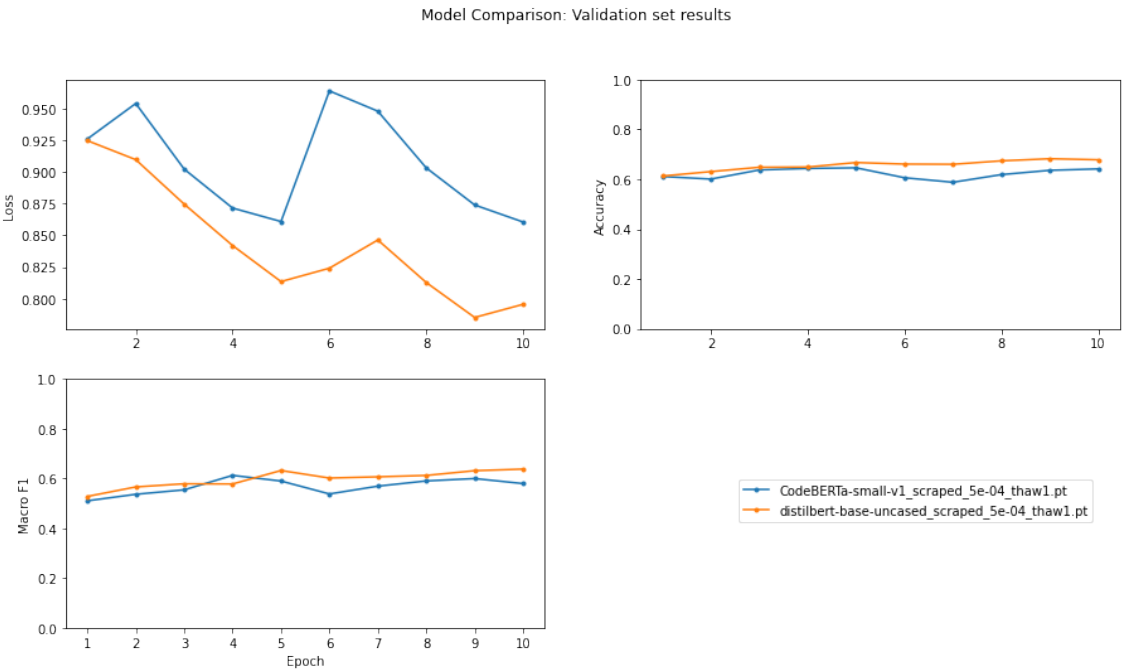


Figure 6.4: Comparison of the best DistilBERT and CodeBERTa versions - trained on extended text data

	Precision	Recall	F1-Score	Support
fix-bugs	0.42	0.83	0.56	76
new-features	0.57	0.49	0.53	149
documentation	0.80	0.71	0.76	52
non-functional	0.60	0.48	0.53	228
accuracy			0.56	505
macro avg	0.60	0.63	0.59	505
weighted avg	0.59	0.56	0.56	505

		Predicted			
		fix-bugs	new-features	documentation	non-functional
True	fix-bugs	0.83	0.04	0.00	0.13
	new-features	0.15	0.49	0.00	0.36
	documentation	0.10	0.02	0.71	0.17
	non-functional	0.25	0.23	0.04	0.48

Table 6.3: Test set metrics for DistilBERT with one unfreezed layer – trained on extended text data.

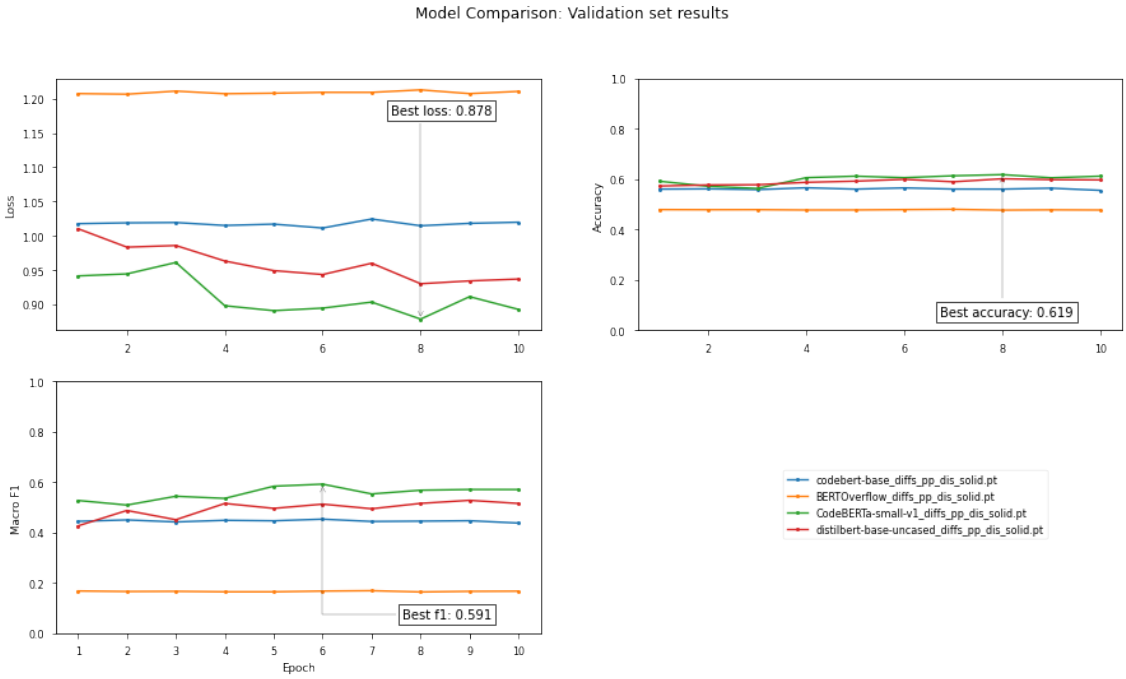


Figure 6.5: Comparison of each transformer with the base configuration - trained on diffs

	Precision	Recall	F1-Score	Support
fix-bugs	0.27	0.76	0.39	76
new-features	0.45	0.69	0.54	149
documentation	0.58	0.27	0.37	52
non-functional	0.62	0.09	0.16	228
accuracy			0.39	505
macro avg	0.48	0.45	0.37	505
weighted avg	0.51	0.39	0.33	505

		Predicted			
		fix-bugs	new-features	documentation	non-functional
True	fix-bugs	0.76	0.22	0.00	0.01
	new-features	0.26	0.69	0.04	0.01
	documentation	0.33	0.19	0.27	0.21
	non-functional	0.46	0.43	0.02	0.09

Table 6.4: Test set metrics for the solid CodeBERTa model – trained on diffs.

	Precision	Recall	F1-Score	Support
fix-bugs	0.46	0.88	0.60	76
new-features	0.56	0.68	0.61	149
documentation	0.85	0.77	0.81	52
non-functional	0.69	0.39	0.50	228
accuracy			0.59	505
macro avg	0.64	0.68	0.63	505
weighted avg	0.63	0.59	0.58	505

		Predicted			
		fix-bugs	new-features	documentation	non-functional
True	fix-bugs	0.88	0.04	0.00	0.08
	new-features	0.11	0.68	0.01	0.20
	documentation	0.13	0.02	0.77	0.08
	non-functional	0.25	0.33	0.03	0.39

Table 6.5: Test set metrics for the ensemble classifier combining the DistilBERT text data model and the CodeBERTa diff model.

Chapter 7
Discussion

Our best model achieves an F1-score of 0.63 on our annotated dataset, which can be interpreted both positively and negatively. Although it's not bad, there is definitely some room for improvement. Due to the systematic differences between training data and test data (measured by inter-annotator agreement, see Sect. 2.3.1) a higher score like 0.9 would practically not have been possible. Considering that, we argue that 0.63 is actually a pretty good result.

It is interesting to note how well DistilBERT did in our experiments and in many cases outperformed BERTOverflow, CodeBERT and CodeBERTa. We would have expected the PL-NL trained embeddings to yield better performance on text from pull requests that often makes use of a code related vocabulary and Markdown[1] formatting. This makes us question the capability of these models to capture the intricate relationship between NL and PL. Like Karmakar and Robbes (2021), we would like to call for future research on pre-training NL-PL transformers.

The size of the input to a transformer is limited by the self-attention operation that scales quadratically with the sequence length. This is the main reason why we had to reduce the input size in our final experiment as described in Sect. 2.4. Beltagy et al. (2020) address this with an alternative self-attention mechanism that scales linearly. Although interesting, it is not something that we have had time to explore in this thesis and we leave it for future work.

[1]The markup language used for formatting text on GitHub, Stackoverflow and other platforms.

Chapter 8
Conclusion

With our best model, we obtained a macro F1 of 0.63 on the classification of pull requests into four categories. We consider our work an important milestone for enabling further research in bridging the gap between code and documentation at Backtick.

8.1 Future Work

This section contains some ideas for future work that we have identified, but did not have time to complete during our thesis.

Hyperparameters. Due to time constraints, we have not spent much time tuning hyperparameters. It would, however, be interesting to find out how much the performance could be improved by optimizing the configurations further. If we had enough time to do this, we would have considered a grid search over all combinations of parameters. This would be very time-consuming and preferably done on a GPU of higher caliber than the one used for our trainings. Some of the hyperparameters with high potential for tuning include learning rate, input composition, number of epochs, number of frozen layers. Even the small amount of tuning we actually did gave significant improvements in certain cases.

Our ensemble classifier essentially uses output concatenation, as described in Sect. 3.6. Another approach would be to have the fully connected layers trainable, including the layers before the networks are merged. This might lead to better results for some number of unfreezed layers.

More Models. Other models and architectures could be explored. We have identified CodeT5 as an interesting candidate, a T5 transformer that has been trained on code (Wang et al., 2021). We would also like to experiment with other neural network architectures. Although transformers generally outperform Bi-LSTM it would be interesting to explore the difference in the domain of code and natural language. Completely new model architectures

are also emerging, with the potential to outperform transformers in new domains. Two of the most interesting and recent advancements in such networks that we have identified are the PALM system from Google (Chowdhery et al., 2022) as well as the Big Science research workshop on large multilingual models and datasets by HuggingFace [1]. These models are trained on multiple natural languages and programming languages. It would be interesting to explore how new models with a better general code and language understanding could improve our results.

Investigate. Surprisingly, our models pre-trained on code do not significantly outperform our models pre-trained on text. This is similar to the results of Karmakar and Robbes (2021) and calls for more investigation.

Create Datasets. A problem that we faced throughout our thesis is the lack of high quality labeled datasets. In order to solve more complicated problems like finding discrepancies between code and documentation, more high quality data is needed. An essential contribution enabling more research on this problem is to create such datasets. This task might be aided by a pull request classifier, since certain types of code changes likely correlate with certain types of changes in the documentation.

We noticed that there may be disagreement between annotators. The IAA between us and DeepRelease were lower than expected. To tackle this challenge, we recommend having multiple overlapping annotators.

8.2 Last Words

Our hope is that the findings and results from this thesis will make it easier to approach the development of more advanced systems that automate even more complex tasks related to the relation between code and its natural language descriptions.

In addition, we hope that our work will facilitate further research leading to more tools for automating tasks which require models with an understanding of code and natural language. Lastly, we hope that the Docks product will benefit from our findings and potentially incorporate new features based on our models.

[1] `https://bigscience.huggingface.co`

References

Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y.-G. (2008). Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, New York, NY, USA. Association for Computing Machinery.

Beltagy, I., Peters, M. E., and Cohan, A. (2020). Longformer: The long-document transformer. *CoRR*, abs/2004.05150.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2016). Enriching word vectors with subword information. *CoRR*, abs/1607.04606.

Chollet, F. (2017). *Deep Learning with Python*. Manning.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. (2022). Palm: Scaling language modeling with pathways. `https://arxiv.org/abs/2204.02311`, [Online; accessed: 2022-05-05].

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Fitch, F. B. (1944). Warren s. mcculloch and walter pitts. a logical calculus of the ideas immanent in nervous activity. bulletin of mathematical biophysics, vol. 5 (1943), pp. 115–133. *Journal of Symbolic Logic*, 9(2):49–50.

Husain, H., Wu, H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

IBM (2020). What are neural networks? `https://www.ibm.com/cloud/learn/neural-networks`, [Online; accessed: 2022-05-09].

IBM (2021). Ibm spss modeler crisp-dm guide. `https://www.ibm.com/docs/en/spss-modeler/SaaS?topic=dm-crisp-help-overview`, [Online; accessed: 2022-05-05].

Jiang, H., Zhu, J., Yang, L., Liang, G., and Zuo, C. (2022). Deeprelease: Language-agnostic release notes generation from pull requests of open-source software. *CoRR*, abs/2201.06720.

Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. (2016). Bag of tricks for efficient text classification. *CoRR*, abs/1607.01759.

Jung, T. (2021). Commitbert: Commit message generation using pre-trained programming language model. *CoRR*, abs/2105.14242.

Karmakar, A. and Robbes, R. (2021). What do pre-trained code models know about code? *CoRR*, abs/2108.11308.

Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Mathis, A., Mamidanna, P., Cury, K. M., Abe, T., Murthy, V. N., Mathis, M. W., and Bethge, M. (2018). Deeplabcut: markerless pose estimation of user-defined body parts with deep learning. *Nature Neuroscience*, 21:1281–1289.

Russell, S. J. and Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Pearson Education.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2019). Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108.

Sun, C., Qiu, X., Xu, Y., and Huang, X. (2019). How to fine-tune BERT for text classification? *CoRR*, abs/1905.05583.

Tabassum, J., Maddela, M., Xu, W., and Ritter, A. (2020). Code and named entity recognition in stackoverflow. In *The Annual Meeting of the Association for Computational Linguistics (ACL)*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA. Curran Associates Inc.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. (2020). Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Yacim, J. and Boshoff, D. (2018). Impact of artificial neural networks training algorithms on accurate prediction of property values. *Journal of Real Estate Research*, 40:375–418.

Yu, S., Xu, L., Zhang, Y., Wu, J., Liao, Z., and Li, Y. (2018). Nbsl: A supervised classification model of pull request in github. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6.

Yu, Y., Yin, G., Wang, T., Yang, C., and Wang, H. (2016). Determinants of pull-based development in the context of continuous integration. *Science China Information Sciences*, 59.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-06-03

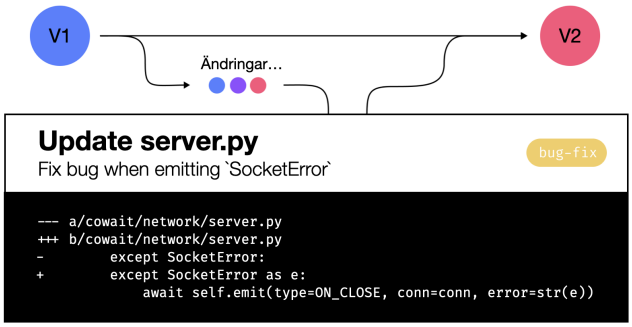
EXAMENSARBETE Classification of Pull Requests using Transformers
STUDENTER Oscar Fridh, Szymon Stypa
HANDLEDARE Pierre Nugues (LTH)
EXAMINATOR Jacek Malec (LTH)

Klassificering av kodändringar som stöd för utvecklare

POPULÄRVETENSKAPLIG SAMMANFATTNING **Oscar Fridh, Szymon Stypa**

Transformer-arkitekturen har gett upphov till kraftfulla modeller för språkförståelse, och börjar dyka upp i nya sammanhang. Detta arbete undersöker hur väl arkitekturen presterar vid klassificering av kodändringar i form av *pull requests*, och är ett försök att förbättra tidigare kategorisering med utökad data bestående av både text och kod.

Mjukvaruprojekt med många utvecklare använder ofta ett arbetssätt där ny funktionalitet utvecklas i separata miljöer för att eliminera konflikter mellan versioner. När en ändring är klar, kan en begäran om att slå ihop dess miljö med centrala projektversionen göras. Denna begäran kallas för *pull request*, och måste godkännas av andra medarbetare för att ändringen ska sammanfogas.



För att underlätta granskningen är det nödvändigt att beskriva alla begäranden, samt kategorisera dem med klasser som tydliggör ändringen. I detta arbete har vi utforskat hur väl det går att automatisera kategoriseringen med hjälp av modeller baserade på *Transformer*-arkitekturen. Målet är att främja nya verktyg som hjälper utvecklare att sköta administrativa uppgifter, och låter dem fokusera på att skriva kod.

I arbetet behandlar vi flera olika förtränade nätverk, och granskar hur de reagerar på olika typer av indata. Genom att använda olika sammanställningar av komponenter från *pull requests* som prediktorer (eng. *features*), studerar vi hur nätverkens förmåga att klassificera påverkas.

Vi demonstrerar att *Transformer*-arkitekturen överlag lämpar sig väl för uppgiften. Slutligen bygger vi en sammansatt modell som uppnår avsevärt bättre resultat än tidigare forksning på vår egeninsamlade data. Samtidigt finns det mycket rum för förbättringar, och framförallt ett behov av mer kvalitativt annoterad data.

Ett oväntat och intressant resultat är att modeller förtränade på vanlig text presterade nästan lika bra eller bättre än modeller förtränade på kod vid klassificering av råa kodändringar. Detta tyder på att det finns ett behov av mer detaljerad forskning kring förträning av modeller, med fokus på hur träningsdata relaterar till modellernas inlärda förståelse.

Vi hoppas att vårt arbete kan agera byggsten för vidare forskning i domänen där kod och språk möter varandra. Automatiserande verktyg för att skriva och underhålla dokumentation är ett exempel där vi tror att arbetet kan komma till nytta.