Classification of Commit Characteristics by Code Changes

MASTER THESIS

Philipp Kramer

Submitted on 2 May 2024

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
M. Sc. Thomas Wolter
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is an original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 2 May 2024

License

Erlangen, 2 May 2024

Abstract

In contemporary software development, Git commits serve as palpable milestones for tracking the progress and evolution of a project. Understanding the types of changes in these commits plays a crucial role in calculating development metrics and improving organizational decision-making. However, even though the benefit of classifying commits is widely recognized, there has been no consistent approach to doing so, especially not in classifying historic commits. In this thesis – using a design science approach – we present a novel approach for automatically classifying commit changes based on the conducted work. This is achieved through the application of a novel machine learning model designed and implemented for this purpose. Furthermore, we developed a widely applicable set of classifications based on previous attempts to classify different types of commits. Both the classifications and the model are constructed using past scientific research and openly available GitHub repositories. In addition, we demonstrate the functionality of our approach by classifying a set of openly available GitHub commits.

Contents

List of Figures

List of Tables

Listings

Acronyms

Adam	Adaptive Moment Estimation
API	Application Programming Interface
CNN	Convolutional Neural Network
ConvLSTM	Convolutional Long Short-Term Memory
CPU	Central Processing Unit
GPU	Graphics Processing Unit
LoC	line of code
LSTM	Long Short-Term Memory
ML	Machine Learning
NLP	Natural Language Processing
ORM	Object Relational Mapping
VCS	Version Control System

1 Introduction

Version Control Systems (VCSs) such as Git represent a foundational aspect of contemporary software development. These systems offer developers a robust platform that facilitates collaboration, codebase integrity, and change tracking. At the core of Git are commits, quantifiable units of changes made to the codebase. Understanding the types of maintenance tasks performed in these commits is of great value to many different stakeholders, including managers as well as software engineers. For both of them, understanding how and where development time is spent, makes "it easier to understand [the] development progress, identify areas that require improvement, and make informed decisions regarding software version releases" (Tong et al., 2023, p. 1). Previous attempts to classify commits, for instance, include the likes of Swanson's *Adaptive*, *Corrective*, and *Perfective* (1976), or de Wilde's *Addition*, *Removal*, *Modification*, *Bugfix*, and *Configuration* (2019).

A substantial amount of previous research has been dedicated to the extraction of information from commit metadata, such as the commit message or the author. In some cases, this data is also combined with high-level source code information, such as the file type or the number of added and deleted lines in each modified file (Hindle et al., 2009; Sarwar et al., 2020). While these approaches achieve promising results within well-defined guidelines, they are susceptible to suboptimal results when faced with certain challenges. For instance, if a commit does not have a commit message, it may well result in an inaccurate classification. As a consequence, our research will focus on source code changes, which are inherent in virtually all commits. Additionally, our approach aspires to address some potential shortcomings of previous work that focuses solely on code changes. These include having a narrow use case of the developed artifact (De Wilde, 2019), or the use of classifications that, in our view, only have limited practical relevance (Levin and Yehudai, 2017).

This thesis aims to bridge existing approaches to classify commits, both from research and from practice, with a novel machine learning (ML) model that aims to automatically classify commits based on their source code changes. Furthermore, we demonstrate the functionality and applicability of the model on a mature, real-world dataset consisting of commits from the open-source repository `lvgl`[1].

In summary, this thesis presents the following contributions:

- A set of classifications that are closely associated with practice.
- A pipeline designed to retrieve commits from a GitHub repository, remove any data not relevant for the classification, and extract the classification label from the commit message.
- A ML model that predicts the classification based on the changes made to the source code in a commit.
- A demonstration that illustrates potential future improvements to our approach.

For this thesis, we chose the design science methodology as our research approach. This choice was made, because, according to Hevner et al., design science is defined with an inherent focus on solving practical and organizational problems with the aim of designing, developing, and evaluating an innovative, tangible artifact (2004). Furthermore, its emphasis on iterative development allows for potential future research to refine previous approaches, further improving artifacts. All of these points are ideally suited to our work, as there is a high organizational relevance in automatically classifying source code changes. In detail, this thesis follows the framework proposed by Peffers et al. (2007). This is reflected in the structure of the thesis as such:

- Firstly, chapter 2 provides an overview of related literature and defines the problem our research intends to solve.
- In chapter 3 we clarify the objectives we set out to achieve with our work.
- Thirdly, in chapter 4 we present the design and theory used for the development of our solution.
- In chapter 5 we provide an overview of the tools and datasets we utilized and lay out the steps we took for the implementation of our solution.
- Chapter 6 serves to demonstrate the functionality and applicability of our solution by analyzing its predictions.
- The objective of chapter 7 is to evaluate the results from chapter 6 against the desired outcomes defined in chapter 3.
- Finally, in chapter 8, we present our concluding thoughts on our research and its results.

[1]https://gitshub.com/lvgl/lvgl/commit/6b1516926a549f9e5a2e07c81a22bb6a33ddddb0

2 Problem Identification

As mentioned in the preceding chapter, classifying commits has a huge potential benefit for decision-makers, such as managers, and developers alike. In this section, we present an introduction to the theory and background of the problem domain and give an overview of previous related work.

2.1 Background

In many contemporary software development projects, having multiple developers collaborating on the same codebase is a common occurrence. This makes VCSs such as Git a crucial tool for managing and tracking changes made to the project. Each of these changes is typically recorded as a commit, which contains information like the modified lines of code (LoCs), a message describing the purpose of the changes, and the author(s). However, as projects evolve, the number of commits steadily grows, making it increasingly difficult to keep track of what and why parts of the codebase changed. As a consequence, there is an ever-increasing need for automated techniques aiding both managers as well as developers in understanding the evolution of the codebase.

2.2 Machine Learning

ML refers to a subfield of artificial intelligence and computer science. Its objective is to design algorithms that enable computers to learn from data and make decisions, without being specifically programmed for a task. This is done by the algorithms recognizing patterns within datasets, which in turn enables them to generalize and forecast previously unseen data. Typically, ML algorithms are categorized either as supervised, unsupervised, or reinforcement learning. Supervised algorithms utilize labeled datasets, meaning the input data has a desired output label assigned to it, which the algorithm is tasked to predict. In contrast, unsupervised learning does not use predefined labels. Rather, the goal of these algorithms is to learn the underlying patterns and group the input data into classes with similar features. Finally, in reinforcement learning, the model learns

given feedback on past outcomes. This process results in a constant feedback loop perpetually improving the performance of the model.
In our thesis, we focus on supervised learning.

2.3 Categorizing Commits

Much of prior research in the area of automated commit classification has focused on the extraction of information from the commit message, often in combination with high-level data about the commits such as the the commit's author(s) or the sheer number of changed lines of code.

A notable approach focusing exclusively on the commit message was proposed by Sarwar et al. (2020). In their work, they employ a pre-trained Natural Language Processing (NLP) model called DistilBERT[1], which they fine-tuned for their use case. For their classifications, they employ the set developed by Swanson (1976): *Adaptive*, *Corrective*, and *Perfective*. A more detailed description of these classifications will be provided in section 4.1. However, unlike previous research in this field, this study allows multi-labeling commits, which means commits may be categorized into several classifications.

Another approach that includes the commit message is by Hindle et al. (2009). Other than Sarwar et al., they subsidize the information obtained from the commit message with high-level features about code changes. These features include the "count of files changed per directory [and the count of] files by their kind of usage: source code, testing, build/configuration management, documentation and other" (Hindle et al., 2009, p. 32). Another difference to Sarwar et al. is the way they analyze the commit message. Instead of using a pre-trained model, they attempt to use a Bayesian-type learner on the frequencies of words in commit messages. For their set of classifications, they use the "Extended Swanson Categories of Changes" (Hindle et al., 2009, p. 31). As the name suggests, they integrated the three classifications developed by Swanson with two additional classifications: *Feature Addition* and *Non-functional*.

Even more sources of data are considered by Casalnuovo et al. (2017). While still utilizing the commit message, the focus of their methodology shifts more to the analysis of source code changes. For the commit message they "convert each commit message to a bag-of-words [(including e.g. *fix, bug, error*)] and then stem them using standard NLP techniques" (Casalnuovo et al., 2017, p. 398). Doing this allows them to identify whether a commit is bug-related or not. What was particularly interesting to us was their approach to analyzing source code

[1]https://huggingface.co/docs/transformers/model_doc/distilbert

changes. It is based on the application of regular expressions, which they for instance utilize to find the beginning and end of functions or code blocks. The proposed extensibility of this approach to other languages led us to consider the idea of classifying multiple languages. However, following preliminary research, it became evident that this approach did not provide sufficient semantic and syntactic information about the modified code. Consequently, the idea of classifying multiple languages was abandoned in favor of focusing on a single one. For more information, see paragraph 4.2.2.

PatchNet is a tool developed by Hoang et al. (2021) that can be used to predict whether or not a commit (to the Linux kernel[2]) is stable or not. To solve this issue, their tool employs a deep learning approach, with a focus on the application of convolutional layers. The convolutional layers are applied to both the analysis of the commit message and at multiple points during the analysis of the source code changes. While the model's classifications were not of interest to our work, the architecture of the model they used for the classification of code changes sparked our interest and influenced our design choices regarding the ML model's architecture.

Another approach that is worth mentioning is that of Levin and Yehudai (2017). In their paper, they present three models: a keyword model for the commit messages, a changes model for source code changes, and a combined model, which merges the other two. All of these models share the same output classifications developed by Swanson. For the analysis of the commit messages, they use word frequency analysis. This methodology enabled the authors to define the ten most common words for each of the three classifications. For the source code analysis, they tested three different machine learning models, including a Random Forest, a Gradient Boosting Machine, and a J48. While their combined model achieved the highest accuracy in their demonstration, the reliance on a commit message could present a challenge in real-world projects. This is supported by a finding made by Hattori and Lanza. According to them, the "frequency [of empty commit messages] is relatively high, even in the case of larger commits" (Hattori and Lanza, 2008, p. 66). This is also the reason why we decided to focus exclusively on the source code changes for our classifier.

To our knowledge, there are only two papers that focus solely on the classification of commits based on source code changes. The first of these is by de Wilde (2019). In his research, he presents the following five classifications: *Addition*, *Removal*, *Modification*, *Fix*, and *Config*. In order to group the code modifications into these classifications, he uses the following approach: If a change only has added/removed LoCs, it is an *Addition*/*Removal*. If a change has both added

[2]https://github.com/torvalds/linux

and removed lines, the Jaro distance between all lines in the changed chunks is calculated. Should two lines have a distance greater than 0.75, they are assumed to be a *Modification*. Should the distance between two lines exceed 0.9, they are considered a *Fix*. Finally, all lines that alter the configuration (e.g. `<script>` tags or `.css` files) are *Config*. It should be noted that, in particular, the classification *Config* is highly specific to HTML-based projects. This narrows the scope of this approach, limiting its adaptability to other languages and projects.

The other paper we found is by Meng et al. (2021). In their research, they present `CClassifier`, a tool used to predict whether a commit is a *Bug fix*, a *Functionality Addition* or if it belongs to *Other* (e.g. refactoring). As they note in their paper, these three classifications are heavily influenced by Swanson's set of classifications. `CClassifier` itself is split into three parts. Firstly, the tool extracts the code changes and gathers information about the relations between changes. Secondly, they model all the extracted data as a Change Dependency Graph (CDG), which illustrates the structural connections between the edited code snippets. Lastly, they pass the graph to a Convolutional Neural Network (CNN) tailored for graph classification.

To summarize, a substantial body of research has been done in the area of automated commit classification. However, only a limited number of studies have been focused on the classification based solely on source code changes. Furthermore, an even smaller body uses classifications that are relevant to general practice. This thesis aims to address this gap in the literature.

3 Objective Definition

As outlined in the previous chapter, there have been several attempts at classifying commits. However, apart from Meng et al. (2021) and de Wilde (2019), none of the related works we found focused solely on semantic and syntactic code changes. The majority of existing work infers the type of commit by analyzing commit messages, log files, and/or high-level information derived from code changes. Furthermore, we discovered a significant discrepancy between the classifications used in research and those used in practice.

Consequently, the objective of this thesis is to bridge the gap between field-proven, practical classifications and the extraction of information from code modifications. We divided this task into six discrete steps, where objective one focuses on the ground rules for the classifications, while points two through six define the fundamental ideas and constraints for the software tool we will develop. This software tool will include a classifier that can be utilized to predict the classification of a commit and a pipeline that gathers information the classifier uses for training, validation, and testing.

Firstly, we identified a number of different approaches used in research to classify commits into maintenance tasks. However, none of these approaches demonstrated great adaptability to practice. Therefore, our first objective is to create a set of classifications that is easily distinguishable, can be used easily for research purposes, and is fully present in most openly available Git repositories, thus ensuring practicability.

Secondly, the software tool's extraction pipeline (hereafter also referred to as the extractor), should be capable of obtaining a list of commits given a GitHub[1] repository. Each of these commits should be filtered for only the information relevant to the classification of said commit. Furthermore, the extractor should also be able to read out the classification from the commit's message.

Thirdly, given a commit, the classifier should be able to output the probabil-

[1]https://github.com/

ities associated with the predefined classifications. Each of our classifications should be assigned a value between 0 and 1 (0-100%), with the sum of all values adding up to 1. This approach allows us to easily identify which classification has the highest probability according to the classifier.

Fourthly, the classifier should be applicable to all repositories, also including those exclusively containing documentation.

Fifthly, we decided that the software tool's classifier should be trained exclusively with data openly available on GitHub.

Lastly, the objective of the classifier is to be able to predict the correct classification (i.e., the one with the highest percentage as defined in objective 5) in at least 80% of all cases.

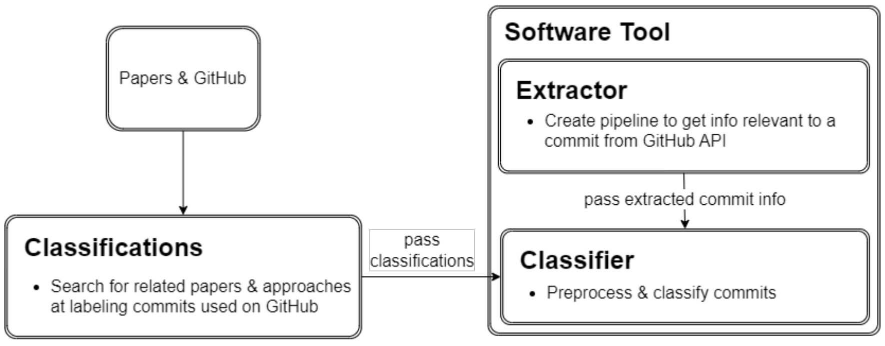
To summarize, the commit classifier by code changes presented in this thesis must fulfill the following criteria:

1. The set of classifications a commit can belong to should be easy to distinguish and should be fully present in most openly available git repositories.
2. The software tool's extractor should, given a repository, extract all information relevant to a commit and extract a tag from the commit message.
3. The software tool's classifier should, given a commit, be able to output the probabilities for the given classifications.
4. The software tool's classifier should be usable on all repositories.
5. The software tool's classifier should be trained exclusively with data openly available in GitHub repositories.
6. The software tool's classifier should be able to predict the correct class at least 80% of the time.

4 Solution Design

As the classifier of our software tool depends on us previously defining the set of classifications the classifier should use, we decided to split our solution process into two major tasks. Firstly, we will analyze a selection of papers and find previous attempts to group commit types into classifications. Additionally, we will examine GitHub in search of any existing schemas in place for differentiating commits.
The second step in our work will focus on the development of the software tool. This tool will incorporate both the extractor and the classifier. Figure 4.1 provides an overview of the process.

Figure 4.1: Overview of the Solution Design Architecture


Papers & GitHub
Classifications
Search for related papers & approaches at labeling commits used on GitHub
pass classifications
Software Tool
Extractor
Create pipeline to get info relevant to a commit from GitHub API
pass extracted commit info
Classifier
Preprocess & classify commits

4.1 Classifications

In order to create a suitable set of classifications, we will consider methods used in both research and practice. For the research-based classifications, we will analyze a selection of papers. Likewise, in order to identify any classification schemas utilized in practice, we will examine a number of repositories on GitHub. Following the analysis, we will establish a set of classifications that fulfills the objectives we set out for them.

4.1.1 Classifications in Research

To find suitable papers describing the approaches to classification, we utilized Google Scholar. After locating a number of potentially relevant literature and classifications using the key search term "classify commits", we proceeded to examine these papers' related work and sources. This process led us to identify three noteworthy methods of classification.

The first method was developed in 1976 by Swanson (1976). In this, the authors identified three distinct types of maintenance tasks: *Adaptive*, *Corrective*, and *Perfective*. *Adaptive* maintenance occurs in response to changes in the data and the runtime environment, for example when new features are created. *Corrective* maintenance refers to tasks performed in response to failures, more commonly called bug fixes. Lastly, work labeled as *Perfective* is performed to combat processing inefficiencies, enhance performance, or increase maintainability. Adapted to our context this includes anything from documentation, tests, and code improvements.

Building on Swanson's method, we identified the "Extended Swanson Categories of Changes" (Hindle et al., 2009). In addition to the three classifications defined by Swanson, Hindle et al. extended it by two more: *Feature Addition* and *Non-functional*. The authors describe *Feature Additions* as dealing with new requirements, which they later clarify as issues related to versioning, merging, or revising. *Non-functional* tasks, in contrast, are described as addressing legal concerns, code clean-up, and source control system management.

The final set of classifications we want to mention was first proposed in Mitch de Wilde's dissertation on "Automatic git commit generation and classification for HTML based projects" (De Wilde, 2019). For his work, he defined the following five classifications: *Addition*, *Removal*, *Modification*, *Bugfix*, and *Configuration*. A commit was labeled as an *Addition* if it exclusively included added lines of code. For *Removal*, the inverse is true. A *Modification* refers to commits that have both added and deleted LoCs. Additionally, this category included file renaming. The label *Bugfix* "was defined as a very small modification such as a typo fix or an added attribute to a HTML tag" (De Wilde, 2019, p. 48). Finally, the classification *Configuration* includes modifications "such as adding external code through `<script>` tags or adding `.css` files" (De Wilde, 2019, p. 26).

In summary, previous scientific work has employed various classification systems, including:

1. *Adaptive*, *Corrective* & *Perfective* by Swanson (Swanson, 1976)
2. *Adaptive*, *Corrective*, *Perfective*, *Feature Addition* & *Non functional* by Hindle et al. (Hindle et al., 2009)
3. *Addition*, *Removal*, *Modification*, *Bugfix* & *Configuration* by de Wilde (De Wilde, 2019)

4.1.2 Classifications in Practice

Since we aimed to define a set of classifications that can be easily applied in a practical context, we also conducted an examination of existing classification schemas on GitHub. All of the classifications presented in the following are a part of the commit message, which is defined by the developer of the corresponding commit.

In our search for suitable GitHub repositories, we initially examined the papers mentioned in subsection 4.1.1 to identify any referenced repositories with classification schemas they had used. The most promising reference we found was in de Wilde's dissertation (2019), where he referred to the *irwin* repository, maintained by Alex Pilewski[1]. The classifications he uses in the repository (up to April 30th, 2024) are:

- *Add*
- *Clean*
- *Create*
- *Delete*
- *Edit*
- *Fix*
- *Initial*
- *Lint*
- *Production*
- *Refactor*
- *Styles*
- *Tests*
- *Update*
- *WIP* (work in progress)

Pilewski does not provide any documentation on the specifics of each classification. This becomes a significant issue when attempting to differentiate between, for instance, the classifications *Styles* and *Lint* or *Add* and *Create*. However, we identified a guide that Pilewski's classifications seem to be related to. This guide can be found in the article "Write categorized Git Commit Messages", published by Ryan Westlake on Medium.com[2] (2017). In this article, Westlake describes a similar approach to Pilewski, but with fewer and slightly different classifications. Their descriptions are as follows:

[1]https://github.com/Pilewski/irwin
[2]https://medium.com/

"

- *Add* — broad category for code that's added
- *Build* — during the build process
- *Fix* — fixes (e.g. typos, linter rules, broken links/imports, etc.)
- *Initial* — for the initial commit and set up
- *Production* — production related
- *Refactor* — refactored code
- *Remove* — when removing files or old, unnecessary code
- *Styles* — style related commits
- *Tests* — if you write them :)
- *Update* — for small updates
- *WIP* — work in progress

" (Westlake, 2017, Categories section). One disadvantage the classifications by Pilewski and Westlake share is that their categories are not mutually exclusive. As Westlake writes, "you can even combine them" (Westlake, 2017, Categories section). While their approaches might be interesting for future research, they are unsuitable for us given the objectives defined prior.

The second approach to finding suitable classifications involved searching for the most popular GitHub repositories and examining whether they utilized a classification schema. For this, we used the key search term "github most popular repositories" on Google[3] and, for instance, identified the following two articles:

- "GitHub's Top 100 Most Valuable Repositories Out of 96 Million" (Gaviar, 2019)
- "15 Most Popular GitHub Repositories Every Developer Should Know" (T., 2023)

Upon examination of the repositories mentioned in these articles, a recurring schema emerged. This schema can be traced back to the Conventional Commits (ConventionalCommits.org, n.d.), or, more specifically, to the *Angular*[4] convention (Angular Team, n.d.). The developers of the *Angular* repository introduced a novel ruleset for the formatting of their commit messages. The objective of this standardization was to improve the readability of the messages and changelog

[3]https://www.google.com/

[4]https://github.com/angular/angular

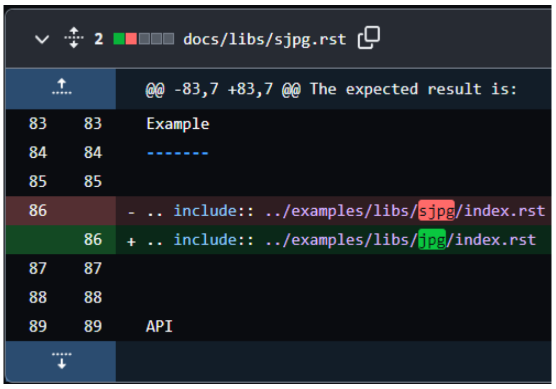
(Angular Team, n.d.). The format of a commit message is as depicted in Listing 4.1:

Listing 4.1: Commit Message Format of the Angular Convention

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

Table 4.1 illustrates the relation between the initial code changes and the resulting commit message.

Table 4.1: Code Change and its Corresponding Commit Message[5]



```
2 docs/libs/sjpg.rst

@@ -83,7 +83,7 @@ The expected result is:
    Example
    -------

  - .. include:: ../examples/libs/sjpg/index.rst
  + .. include:: ../examples/libs/jpg/index.rst


    API
```

→ docs(jpg): fix example path

Of the message format, the header "`<type>(<scope>): <subject>`" or more precisely the `<type>` (in the example *docs*), piqued our interest. *Angular's* types are defined as follows:

- *build*
- *ci*
- *docs*
- *feat*
- *fix*
- *perf*
- *refactor*
- *revert*
- *style*
- *test*

Building on these types and more generally on the *Angular* convention, Conventional Commits (ConventionalCommits.org, n.d.) were introduced. Along with some additional semantic modifications, e.g. for breaking changes, Conventional Commits added the new type *chore*. In summary, the classifications of Conventional Commits are described as such:

[5]https://github.com/lvgl/lvgl/commit/6b1516926a549f9e5a2e07c81a22bb6a33ddddb0

- *build* — Changes that affect the build system or external dependencies
- *chore* — Updating grunt tasks etc; no production code changes
- *ci* — Changes to our CI configuration files and scripts
- *docs* — Documentation only changes
- *feat* — A new feature
- *fix* — A bug fix
- *perf* — A code change that improves performance
- *refactor* — A code change that neither fixes a bug nor adds a feature
- *revert* — If the commit reverts a previous [list of] commit
- *style* — Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc)
- *test* — Adding missing tests or correcting existing tests

The descriptions of the classifications are derived from (Ziegelmayer, n.d.) for "*chore*" and from (Angular Team, n.d.) for the other classifications. Unlike Pilewski's or Westlake's classifications, Conventional Commits' are designed to be mutually exclusive. This fact allows developers writing commit messages using this schema to unambiguously assign a classification to their changes. Once we knew that Conventional Commits existed, we utilized the built-in search function of GitHub to identify repositories using this schema. For this, we utilized this search term:

```
https://github.com/search?q="conventional+commits"++path%3A**
%2FCONTRIBUTING.md&type=code&ref=advsearch
```

. This search term looks for repositories that contain a file with the name `CONTRIBUTING.md`, in which the string "conventional commits" is mentioned. This way we found the repositories mentioned in section 5.2.

4.1.3 Our Classifications

As outlined above, there are various different ways of categorizing commits into maintenance tasks. Some methods, such as Pilewski's or Westlake's, are not suitable for our approach, because their classifications are not designed to be mutually exclusive. Others, like de Wilde's (2019), proposed classifications for a specific context (in his case HTML), which defeats the purpose of having a widely usable set of classifications. Ultimately, we opted for a slight variation to the Conventional Commits:

- *build*
- *chore*
- *ci*
- *docs*
- *feat*
- *fix*
- *perf*
- *refactor*
- *style*
- *test*

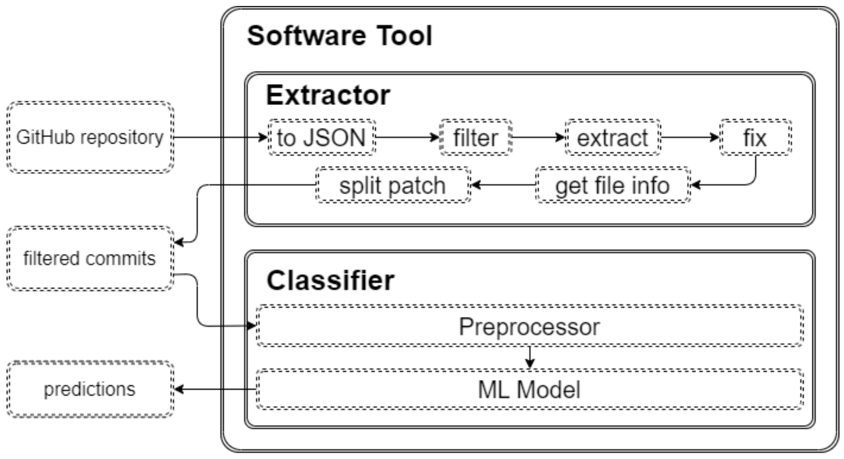
The rationale behind the selection of this set of classifications can be explained by three main reasons:

- Firstly, Conventional Commits are widely adopted in practice, fulfilling all the objectives we set for the classifications.
- Secondly, since they are already widely used, we can use the adhering repositories as pre-labeled datasets for the training of the classifier, eliminating the need for the manual creation of such data.
- Lastly, we decided to remove the classification *revert* because a substantial amount of repositories label their revert commits as *chore*, which would present a great challenge if we wanted to separate these commits again.

4.2 Software Tool

Having established the set of classifications, we will in the following proceed to describe the components of the software tool. As illustrated in Figure 4.2, the software tool is not one single entity but consists of multiple different modules. First, we will describe the extractor, explaining each part of the extraction pipeline in a step-by-step manner, illustrating how the tool transforms a list of commits into a dataset that can be used by the classifier for training and evaluation. Afterward, we will provide a detailed description of our classifier, describing its components and the rationale behind the design of the preprocessor and the novel ML model.

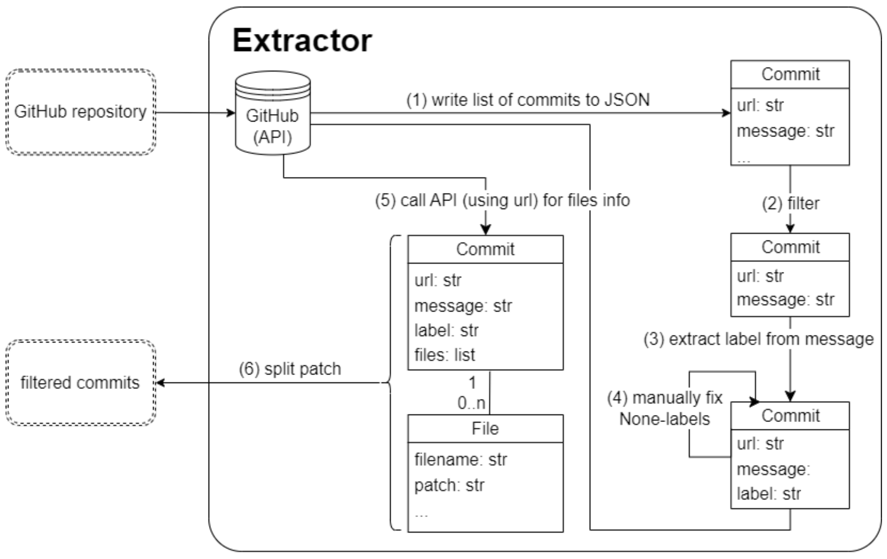
Figure 4.2: Overview of the Software Tool



4.2.1 Extractor

Starting with the extractor. The extractor is composed of six subcomponents, as Figure 4.2 implies and Figure 4.3 illustrates. In the following, we will provide a detailed explanation of these six steps.

Figure 4.3: Architecture and Steps of the Extractor Pipeline



In (1) we make the initial call to the GitHub Application Programming Interface (API), requesting a list of commits in chronologically descending order. To ensure consistency in the number of commits across all repositories in our dataset, we added an attribute limiting the number of commits to a specified value. Furthermore, we also write this list of commits to a JSON file. This approach allows us to reuse the commit data in the JSON file for the subsequent steps, thus avoiding the need to make API calls repeatedly. Consequently, this prevents the unnecessary exhaustion of our GitHub API quota.

In step (2), we then proceed to iterate through each commit in the output file from (1) and filter out any unimportant attributes. At the conclusion of this process, we are left with two attributes: the commit's `url` and its `commit message`. The `url` is used to uniquely identify each commit, while the `commit message` contains the classification that we will extract in step (3). Initially, we considered utilizing the commit's `sha` as the primary key. However, since our first approach for the classifier involved classifying code changes from multiple languages and thus repositories, the `sha` might not be distinct across all these repositories, which would render it an inadequate identifier.

As mentioned above, in step (3), the program extracts the classification label from the commit message. This is achieved by utilizing a regular expression in order to split the message into several groups. Of these, the first group is the most interesting for us, as it contains the `<type>` assigned to the commit. This value is then matched against the list of classifications we previously established (see subsection 4.1.3). If a match is found, the commit's classification is defined as the `<type>`. In case no match is identified initially, the tool performs an additional check for some special cases we encountered in our selected repositories. For instance, one repository used the label `doc` instead of `docs` for their documentation-related modifications. Consequently, it can be concluded that when the `<type>` `doc` is encountered, the tool can set the commit's classification to `docs`. Should, even after considering these special cases, no match be found, the commit's classification is temporarily set to `None`, indicating it will need to be manually fixed in step (4).

In step (4), we then consider all commits that have received a `None`-label. This is the part of the pipeline that requires manual work and is, therefore, the most time-consuming. A small portion of these commits can be fixed easily since their commit message entailed one of the defined classifications, but the label was misspelled. However, the majority of the `None`-classified commits require additional work. For each commit, we examined the commit message and determined which classification was indicated by it. If there were still ambiguities about which classification to choose, we reviewed the diff file in order to assess the actual code changes.

Utilizing the `url` as the identifier for the commits provides the additional benefit that we can reuse it for the second call of the GitHub API in step (5). To illustrate why this is possible, consider the following example `url` obtained in step (1):

```
https://api.github.com/repos/lvgl/lvgl/commits/
fa9142ef361f548c534f5e1d2144f94c88b3873e
```

Upon comparing the `url` with the structure required by the GitHub API for the `"Get a commit"` call

```
/repos/{owner}/{repo}/commits/{ref}
```

, it becomes evident that the `url` can be reused to obtain the rest of the remaining information required for the classifier. This additional information includes details about the files that were modified in each commit. More specifically, it includes the number of LoCs added and deleted, the filenames, as well as the patch that specifies the exact modifications made in each file. Using this additional data we extend the previous dictionary by the new values and pass it on to step (6).

In the $(6)^{th}$ and final step, the lines of code in each patch are divided into three categories:

- If a line starts with "+", it is considered an added line.
- If a line starts with "-", it is a deleted line.
- If a line starts with neither "+" nor "-", the line is ignored by our extractor.

We decided to ignore all lines neither considered added nor deleted because including them would have significantly inflated the size of the dataset without providing a substantial increase in information. To illustrate the dividing process, consider the following example illustrated in Listing 4.2:

Listing 4.2: Example of an Unformatted Patch

```
"@@ -56,6 +56,7 @@ void _lv_indev_scroll_handler(
   lv_indev_t * indev)\n \n
   init_scroll_limits(indev);\n \n+
   lv_obj_remove_state(indev->pointer.act_obj,
   LV_STATE_PRESSED);\n         lv_obj_send_event(
   scroll_obj, LV_EVENT_SCROLL_BEGIN, NULL);\n
           if(indev->reset_query) return;\n     }"
```

A reformatted version of this patch to make it more human-readable is presented in Listing 4.3.

Listing 4.3: Example of a Human-Readable Patch

```
@@ -56,6 +56,7 @@ void _lv_indev_scroll_handler(
   lv_indev_t * indev)

     init_scroll_limits(indev);

+    lv_obj_remove_state(indev->pointer.act_obj,
   LV_STATE_PRESSED);
     lv_obj_send_event(scroll_obj,
        LV_EVENT_SCROLL_BEGIN, NULL);
     if(indev->reset_query) return;
}
```

In the presented patch, the line

```
lv_obj_remove_state(indev->pointer.act_obj, LV_STATE_PRESSED);
```

was added and no line of code was deleted. In addition to splitting the LoCs into the three aforementioned sets, we also intended to retain as much information as possible about the context and structure of these lines without overly inflating the dataset. For that, we introduced an additional dimension, which we refer to as *chunks*. If two lines of the same category (added/deleted) are adjacent to each other, they are added to the same chunk. For instance, if a new file is added to the repository, the entire file would belong to the class "added", with all lines of code being included in the same chunk.

Upon completion of the extractor's pipeline, a commit in the output file would, for instance, look like Listing 4.4.

Listing 4.4: Example Output of an Extracted Commit

```
{
    "url":"https:\/\/api.github.com\/repos\/lvgl\/
       lvgl\/commits\/
       fa9142ef361f548c534f5e1d2144f94c88b3873e",
    "tag":"feat",
    "filename":[
        "src\/indev\/lv_indev_scroll.c"
    ],
    "file_extension":[
        "c"
    ],
    "added_patch":[
        [
            [
                "        lv_obj_remove_state(indev
                   ->pointer.act_obj,
                   LV_STATE_PRESSED);"
            ]
        ]
    ],
    "deleted_patch":[
        [

        ]
    ],
    "status":[
        "modified"
    ],
    "additions":[
        1
    ],
    "deletions":[
        0
    ]
}
```

4.2.2 Classifier

Once all relevant information for the classification of the commits has been extracted, the output file is passed on to the classifier. Due to the thesis's time constraints, it was decided to focus on developing a model that exclusively utilizes the attributes `added_patch` and `deleted_patch` as inputs and classify the commit based on them. These two attributes were selected, because they provide the core information about the commits' source code changes, which is to say that they have the highest information density of all attributes. Furthermore, by focusing on these two, we can create a model that processes solely textual data, eliminating the need for additional submodels to handle other data types. Consequently, the schema for each commit is as follows:

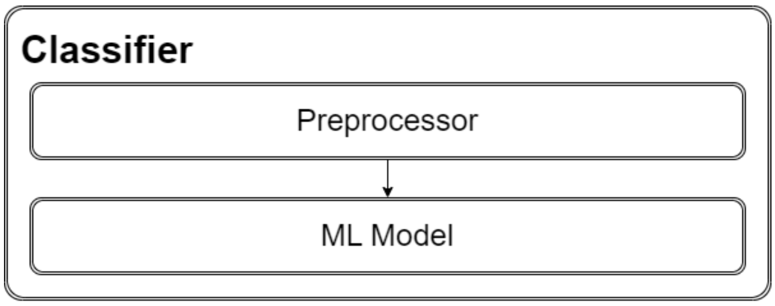
Listing 4.5: Schema of Each Commit Before the Preprocessor

```
{
    "tag": "string",
    "added_patch": "list(list(string))",
    "deleted_patch": "list(list(string))"
}
```

As can be seen in Listing 4.5, in addition to the attributes related to the patch, we have retained the `tag` (also referred to as classification). This is because it is the expected classification used during supervised training.

As depicted in Figure 4.4, the classifier is divided into two primary components. The first component is the preprocessor, a pipeline that receives the added and deleted patches from the extractor and processes them through a series of transformations in order to produce a numeric representation of the inputs. This numeric representation is essential, because the second component, the ML model, is only able to process numbers. In the following section, we will provide a more detailed account of the approaches we chose for both.

Figure 4.4: Overview of the Classifier Architecture



Preprocessors

As previously stated, a preprocessor is a crucial component of the data pipeline. It performs one or more transformations on the raw input data, converting the initial data type, whether it is textual, categorical, or otherwise, into a numeric representation. In the context of textual data, one of these steps must be text

vectorization, i.e. converting words (and, on occasion, punctuation marks) into numeric values.
During the course of our research, we trialed three different preprocessor configurations. A detailed explanation of each approach is provided in the following paragraphs.

Preprocessor 1 The first preprocessor is a relatively basic preprocessor for textual data. Figure 4.5 depicts that it accepts a line of code as input, passes this line through to a `TextVectorization` layer, and outputs the numerical representation of the words in the line of code. In order to use a `TextVectorization` layer, it is initially necessary to provide it with the words present in the dataset. Using these words, the layer is able to learn the vocabulary specific to the task. To prevent rare words from negatively affecting the model's performance, we tested several different vocabulary sizes, with values of 1,000, 5,000, and 10,000. Any words that are too uncommon to be included in the vocabulary are then grouped in the token '`[UNK]`'. Furthermore, given that we initially considered classifying multiple programming languages, the idea of this preprocessor was to normalize all of these languages by eliminating any punctuation. To achieve this, we used the standardization technique `lower_and_strip_punctuation` provided by `TensorFlow`'s `TextVectorization`[6]. In addition to stripping the punctuation, this method also lower-cases all words, i.e. "standardizeText" would be transformed to "standardizetext".

After setting up the `TextVectorization` and adapting it to a vocabulary, `TensorFlow` (Martín Abadi et al., 2015) describes the vectorization steps as follows: "

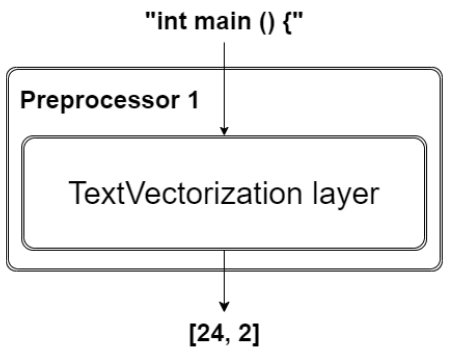
1. Standardize each example (usually lowercasing + punctuation stripping)
2. Split each example into substrings (usually words)
3. Recombine substrings into tokens (usually ngrams)
4. Index tokens (associate a unique int value with each token)
5. Transform each example using this index, either into a vector of ints or a dense float vector.

" (Google Brain Team, n.d.-b, Used in the notebooks section).

To illustrate a potential outcome of applying this preprocessor, consider the example depicted in Figure 4.5. Given an input of `int main () {`, `int` might be vectorized to `24`, `main` to `2` and all punctuation is ignored.

[6]https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization

Figure 4.5: Architecture and Example of Preprocessor 1



Preprocessor 2 Preprocessor 2 reintroduces the punctuation marks that were excluded in preprocessor 1. Upon recognizing that the classifier's ML model did not learn patterns without the context provided by punctuation, the next logical step was to retain it. To achieve this, we tested three different methods:

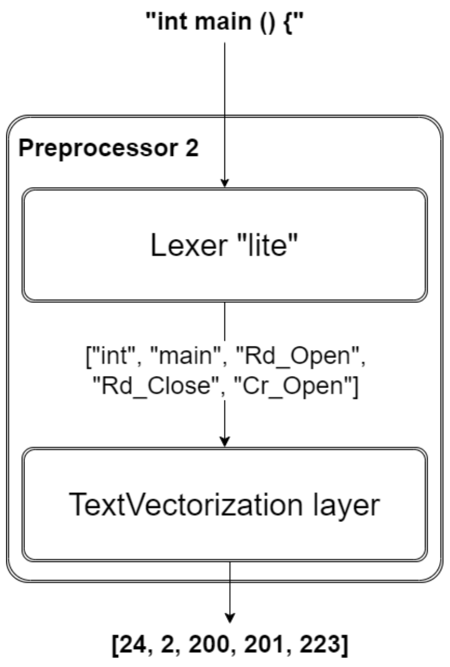
1. Using `TextVectorization's` built-in standardization `lower`
2. Using `TextVectorization's` built-in standardization `None`
3. Manually changing the punctuations to word-tokens (e.g. `"("` → `"Rd_Open"`)

Methods 1. and 2. do not alter the structure of the preprocessor in comparison to preprocessor 1, they merely change the standardization method. In 1., we maintain lowercasing of the input text (in accordance with preprocessor 1), while retaining the punctuation marks. In method 2., however, no standardization is applied. This approach essentially works like a pass-through to the subsequent steps in the vectorization process. As a third option, we elected to manually translate all punctuation marks to text tokens and then pass them to a `TextVectorization` layer that standardizes using `lower_and_strip_punctuation`. The tokens we used can be found in appendix A. This is also the method illustrated in Figure 4.6. Ultimately, the manual method yields the same output as method 1., but it offers a more precise grip on which punctuation marks to include and exclude.

As illustrated by the example in Figure 4.6, the retention of punctuation allows the model to work with more information about the input text. Taking a look at the same example as above, `int main () {` this time not only converts `int` → `24`, `main` → `2`, but also the brackets (here `"("` → `200`, `")"` → `201` and `"{"` → `223`).

Preprocessor 3 After recognizing that the more general, mainly language-independent approaches proposed in preprocessors 1 and 2 yielded no suitable abstraction of the input data, we made two decisions:

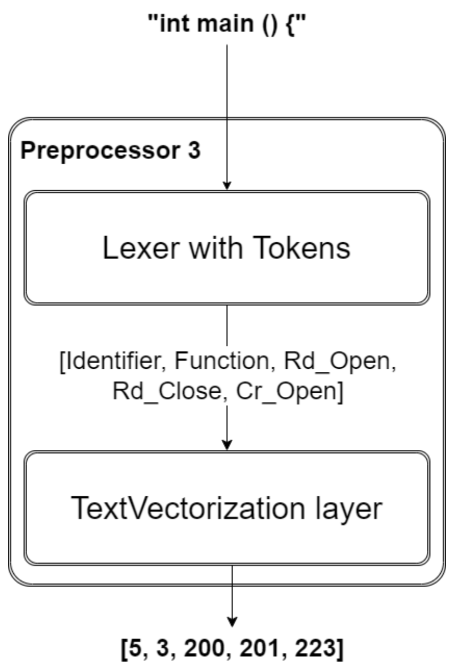
Figure 4.6: Architecture and Example of Preprocessor 2



- Given that we have thus far attempted to preprocess and classify a multitude of programming languages, we elected to focus on a single language.
- In order to vectorize a single language, one of the most effective methods is to utilize a lexer.

The programming language we elected to use is C. This choice was made because C's instruction set, and thus lexer, is relatively small in comparison to other, more high-level programming languages. Additionally, we had already utilized a repository with a codebase written in C, which had a significant number of commits, thus providing a suitably large dataset. This allowed us to use a relatively compact lexer as a step in our preprocessor 3. To gain a better understanding of the role of a lexer, consider the following example: In our previous two preprocessors, a line such as "`int main () {`" would be "translated" to "`[int, main]`" and "`[int, main, Rd_Open, Rd_Close, Cr_Open]`" respectively. However, by using a lexer, keywords such as `int`, `float` or `void` would for instance be grouped under the token `Identifier`, while all function names would be summarized under the token `Function`, and so on. The use of these tokens facilitates the recognition of the code's structure, as shown in Figure 4.7. The program then attempts to deduce a sort of grammar/syntax for the source code, similar to that of natural languages such as English: "Subject - Predicate - Object". Thus, by using a lexer, the objective is to provide a syntax-focused representation of the modified code, with the intention that the classifier will perform more effectively using this method.

Figure 4.7: Architecture and Example of Preprocessor 3



Ultimately, regardless of which demonstrated preprocessors is used, the output schema is the same (see Listing 4.6):

Listing 4.6: Schema of Each Commit After the Preprocessor

```
{
    "tag": "string",
    "added_patch": "list(list(list(int)))",
    "deleted_patch": "list(list(list(int)))"
}
```

However, the primary distinction is that each LoC is now represented by a list of integers instead of a string. To illustrate, one such line could now be (see Listing 4.7):

Listing 4.7: Example Output of a Preprocessed Patch

```
[490, 200, 982, 269, 98, 140, 632, 872, 201, 55]
% output of "lv_obj_remove_state(indev->pointer.
   act_obj, LV_STATE_PRESSED);"
```

Prior to illustrating an example following the application of the preprocessor's steps, it is necessary to mention one additional step we added to the preprocessing pipeline after some preliminary research: padding/truncating. As can be expected, the length of the lines of code varies considerably across the data. In order to address this issue, we initially attempted to use so-called `ragged tensors`. These types of tensors permit the saving and working with data of inconsistent dimensionalities. However, after some research, it became evident that the use of ragged tensors and their undefined dimensionalities would not allow the usage of the ML model layers we planned to use. Consequently, we added the aforementioned padding. To explain the functionality of the padding step, consider the following scenario: Imagine we used a padding token of `-1` and fixed the dimension at `5`. In this case, all lines of code would undergo one of the following transformations:

- < 5: e.g. $[534, 200, 201] \rightarrow [534, 200, 201, -1, -1]$
- $== 5$: e.g. $[45, 200, 431, 10, 201] \rightarrow [45, 200, 431, 10, 201]$
- > 5: e.g. $[234, 564, 239, 214, 981, 234, 345] \rightarrow [234, 564, 239, 214, 981]$

Ultimately, the output of the preprocessor given an input dataset with one commit might look like Listing 4.8:

Listing 4.8: Example of a Dataset With One Commit

```
[
    {
        "tag":"feat",
        "added_patch":[
            [
                [
                    [490, 200, 982, 269, 98]
                ],
                [
                    [326, 21, 2, 424, 564]
                ]
            ],
            [
                [
                    [-1, -1, -1, -1, -1]
                ],
                [
                    [-1, -1, -1, -1, -1]
                ]
            ]
        ],
        "deleted_patch":[
            [
                [
                    [245, 842, 437, 200, 201]
                ],
                [
                    [-1, -1, -1, -1, -1]
                ]
            ],
                    [
                        [
                            [316, 324, -1, -1, -1]
                        ],
                        [
                            [-1, -1, -1, -1, -1]
                        ]
                    ]
                ]
            }
        ]
```

In the case presented above, the commit's classification/`tag` is `feat`. Furthermore, the commit's dimensionalities have been defined as `[1, 2, 2, 1, 5]` for both `added_patch` and `deleted_patch`. As can be seen, the provided dimensionality list has five numbers. These dimensions are:

- c commits (1)
- f files (2)
- ch chunks (2)
- l lines (1)
- w words (5)

While fixing the initial problem of compatibility, the use of fixed dimensions introduces two potential limitations. Firstly, if the dimensionality is set too low, the model may lack the crucial information required to learn patterns in the input. On the contrary, if the dimensionality is set too high, a significant number of padding tokens may be introduced. This, in turn, may result in the model attempting to learn patterns from the padding, even though the paddings are merely a means of norming the input.

ML Models

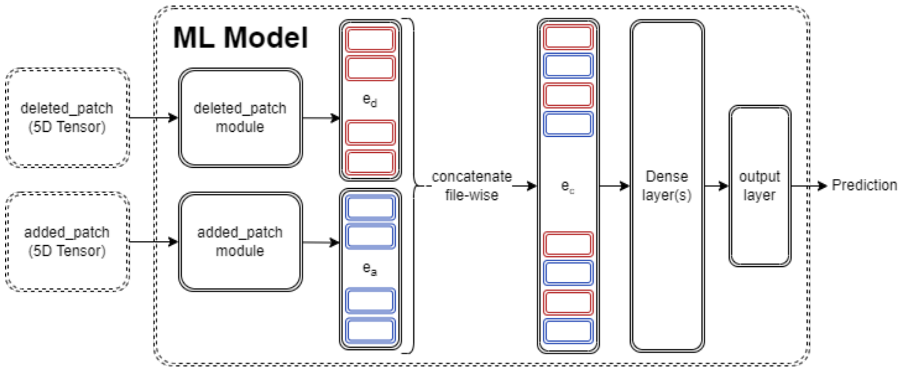
After extracting the training dataset and preprocessing it to ensure compatibility with the ML model, we will in the following describe the model itself. During the course of our work, we developed two different machine learning models for the classification. As illustrated in Figure 4.8, both models share the same general architecture, with the difference between them being the manner in which they handle the commits' patches. Furthermore, both models are tasked with the same learning task: Construct a function

$$f(inputs) \mapsto prediction$$

, where

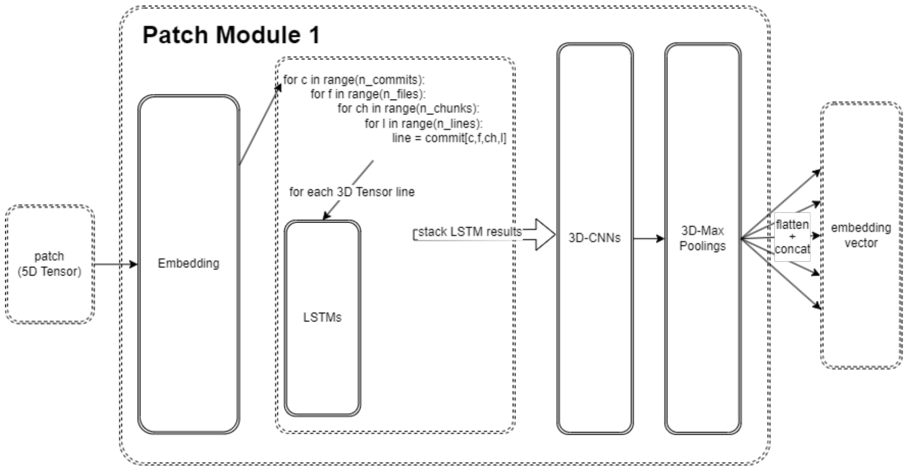
$$prediction \in [build, chore, ci, docs, feat, fix, perf, refactor, style, test]$$

Figure 4.8: Architecture of the ML Model



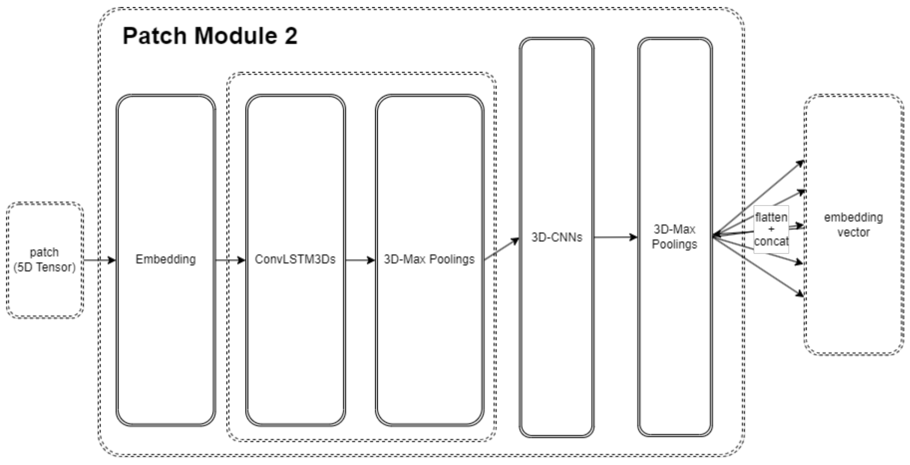
As input for the ML model, we pass the `deleted_patch` and `added_patch`. These inputs are then passed to the patch modules, the part of the model that differs in the two versions. The inputs are then processed in the *(added/deleted) patch modules*. These modules are the components of the ML models that differ in our two versions. Following the patch modules, the resulting one-dimensional embedding vectors e_d and e_a are concatenated file-wise, such that the resulting embedding e_c has alternating added and deleted blocks. This is done in order to keep the structural information of each commit's patch. After the concatenation, a set of Dense layers is applied, with the final Dense layer (`output layer`) having an output dimension of 10 (the number of classifications). Each of the ten output values in the prediction represents one of the classifications. Furthermore, each value is between 0 and 1, with the sum of all values being 1. This indicates that the resulting values each represent the probability for each of the ten classifications.

Figure 4.9: Architecture of Patch Module 1


```
for c in range(n_commits):
    for f in range(n_files):
        for ch in range(n_chunks):
            for l in range(n_lines):
                line = commit[c,f,ch,l]
```

Patch Module 1 Figure 4.9 shows the structure of the first patch module. It should be noted that the architecture depicted here is identical for both the added and deleted patch modules. We elected to mention them separately in Figure 4.8 to provide clarity that they do neither share weights nor information, they are merely designed similarly. As previously mentioned, the patch module gets the five-dimensional patch information. In the first step, the dimensionality of the patch is expanded from five to six through the use of an embedding layer. This is done to improve the contextualization of the code changes, thereby enabling the model to better understand the meaning of the words in the LoCs. Secondly, for each of the lines in the modified code, a set of Long Short-Term Memory (LSTM) layers is applied. Each line at this point consists of three dimensions: the lines-dimension, the words-dimension, and the embeddings-dimension. The rationale behind using LSTM layers is to capture sequential dependencies in the lines, thereby further improving contextualization. The results for each processed LoCs are then stacked in order to regain the six-dimensional structure of the patch. Subsequently, a set of three-dimensional convolutional layers is utilized, with the output of which being passed to a list of 3D-Max Pooling layers. These two sets of layers are designed to gather additional information about the structure of each patch. Lastly, the output of the pooling layers is then flattened to a single dimension and concatenated such that we receive the embedding vector e_a/e_d.

Figure 4.10: Architecture of Patch Module 2



Patch Module 2 The architecture of Patch module 2 is depicted in Figure 4.10. As can be observed, both the embedding layer at the start and the layers at the end are identical to those of patch module 1. However, the way in which they process the lines of code is different. In module 1, each line of code is passed to a list of LSTM layers, each of which internally computes matrix multiplications. Afterward, the result of these layers is stacked to restore the patch's structure. In contrast, in module 2, the matrix multiplications inside the LSTM cell are replaced with convolutional operations. In addition to the method change potentially improving the predictions, this approach has an additional benefit. Since the Convolutional Long Short-Term Memory (ConvLSTM) developed by Shi et al. (2015) allows for six-dimensional inputs, there is neither the need for stacking the LSTM results nor for the custom implementation to access the LoCs.

5 Implementation

This section outlines the implementation of the solution described in chapter 4. In the first step, we will discuss the tools and their contributions to our research. Secondly, we will outline the data sources we worked with. Lastly, we will describe some of the key code snippets used in the software tool.

5.1 Used Tools

To implement our solution, we utilized several different software tools. Firstly, we used `Docker`[1], a platform for containerizing applications, to create the development environments. Additionally, we utilized `Jupyter Notebook`[2] and its next-generational version `JupyterLab`[3] for the entirety of the software tool's development. Both tools are interactive computing environments widely used for data analysis, machine learning, and other purposes. To simplify the handling of the commit-file relations during the extraction process, we used `peewee`[4], an Object Relational Mapping (ORM), in combination with a PostgreSQL[5] database. Finally, for the purpose of developing the classifier, we utilized three additional tools: `Google Colab`[6], a `Jupyter Notebook` service by Google, `Pygments`[7], a generic syntax highlighter, for the Lexer in preprocessor 3, and `TensorFlow`[8], an open-source machine learning platform, for both the preprocessor and the ML model.

[1]https://www.docker.com/

[2]https://jupyter.org/

[3]https://jupyter.org/

[4]https://docs.peewee-orm.com/en/latest/

[5]https://www.postgresql.org/

[6]https://colab.google/

[7]https://pygments.org/

[8]https://www.tensorflow.org/

5.1.1 Docker

In order to ensure a seamless development experience, we opted to stay away from local development and instead implemented everything in two `Docker` development containers. `Docker` itself is "an open platform for developing, shipping, and running applications [and it] enables you to separate your applications from your infrastructure" (Docker Inc., n.d., para. 1). We decided to split our work into two containers because this allowed us to use two separate `Docker` images. The first image was configured using Microsoft's `Dev Containers` extension[9] for Visual Studio Code[10]. It is designed to run code written in Python and provides a connection to a PostgreSQL database. The second image is `gpu-jupyter`[11]. It is developed and maintained by iot-salzburg and it is "optimized to run [. . .] TensorFlow [. . .] in collaborative notebooks on the [Nvidia] GPU" (iot-salzburg, n.d., About section). Using this image enabled us to execute the classifier's code on our local Nvidia Graphics Processing Unit (GPU) using a `JupyterLab` web environment. To facilitate the process of starting and accessing the server, we developed a short `bat` file to automate the execution of the required `Docker` commands. The file's contents are depicted in Listing 5.1:

Listing 5.1: `Bat` File Used to Automate Starting the `JupyterLab` Server

```
cd \Projects
docker start 3c68bc67bcaf
docker exec -it 3c68bc67bcaf jupyter server list
pause >nul
```

In the first line, the working directory in changed to `Projects`, a local folder. Afterward, the `Docker` container with the id `3c68bc67bcaf` containing the `gpu-jupyter` image is started. In line three, the command "`jupyter server list`" is executed on the now-running container. This command returns the list of currently running Jupyter servers. These servers are accessible via the web browser at `localhost:8848`. The fourth and final line is a workaround to ensure that the terminal window in which the `bat` file is executed does not close immediately.

5.1.2 Jupyter Notebook/JupyterLab

`Jupyter Notebook` (and its further development `JupyterLab`), formerly called IPython Notebook, is an open-source "notebook authoring application and is part of Project Jupyter[12]"(Jupyter Team, n.d., para. 1). Both interfaces permit the

[9]https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-containers
[10]https://code.visualstudio.com/
[11]https://github.com/iot-salzburg/gpu-jupyter
[12]https://docs.jupyter.org/en/latest/

execution of several programming languages, including R[13], Go[14] and Python[15]. The main advantage of using these notebook interfaces is the ability to edit and execute code cells without the need to recompile the entire file. This is possible because each notebook file maintains a record track of variables throughout the entirety of the file. This became especially useful in the classifier because it allowed us to retrain the ML model without having to rerun the preprocessing steps. This saves both time and resources during the development process. To use the notebooks, we utilized two methods:

- In method one, the `Docker` image `gpu-jupyter` was employed. This image enabled us to work on a self-hosted `JupyterLab` web server.
- In method two, we utilized the `Jupyter` extension[16], available via the Visual Studio Code Marketplace. This extension enabled us to use a full-fledged `Jupyter Notebook` environment inside Visual Studio Code.

5.1.3 peewee

Given the one-to-many relation between commits and files, we determined that a database was the best medium for modeling this relationship Consequently, we sought an effective and user-friendly solution for reading and writing this database. Our search led us to `peewee`, a "simple and small ORM" (Leifer, n.d., para. 1), brought to life by Charles Leifer in October 2010. Object-relational mapping (ORM) is a technique that enables querying and manipulating data from a database in object-oriented programming languages such as Python. It is a design pattern that aims to streamline the communication between the two parts of the program. The manner in which we employed `peewee` can be observed in subsection 5.3.3.

5.1.4 Google Colab

As previously mentioned, `Google Colab` is a hosted `Jupyter Notebook` service that provides a free computing environment with access to GPUs. Since it provides access to these GPUs, it is particularly well suited for machine learning tasks. However, one partial drawback of `Google Colab` became evident after a brief period of working with it: it has a GPU quota. Given this issue, we had to decide whether to continue training the ML model on `Google Colab`, where we would have to intermittently train it using a Central Processing Unit (CPU), or to switch to a local environment using our local CPU. After conducting speed

[13]https://www.r-project.org/about.html
[14]https://go.dev/
[15]https://www.python.org/
[16]https://marketplace.visualstudio.com/items?itemName=ms-toolsai.jupyter

tests, we ultimately opted to switch to using the local GPU. For this, we used the previously described `gpu-jupyter` image, provided by iot-salzburg.

5.1.5 Pygments

`Pygments` is a syntax highlighting library written in Python. For the syntactic highlighting we utilized in preprocessor 3, it offers 74 distinct token types. The output of highlighting text is provided by `Pygments` in multiple different output formats, including formats such as HTML, LaTeX, and most importantly for our use case, as a list. Detailed information regarding the use of `Pygments` can be found in subsection 5.3.5.

5.1.6 TensorFlow

`TensorFlow` is an end-to-end open source machine learning platform (Google Brain Team, n.d.-a) developed by the Google Brain Team[17]. It provides a comprehensive ecosystem of libraries and resources that facilitate the development of neural networks of any size and complexity, while ensuring flexibility and scalability. For instance, it provides ready-to-use layers, such as the aforementioned `TextVectorization` layer, as well as more complex ones like `ConvLSTM`[18] (Shi et al., 2015). Additionally, it provides an intuitive syntax for implementing, training, and evaluating ML models, which we will discuss in greater detail in section 5.3. In addition to being an invaluable tool for the development of machine-learning tasks, `Tensorflow` furthermore provides a highly compatible interface with other useful tools used in the ML context. Three of these tools were employed in our research. Firstly, we used the library `pandas`[19] in order to facilitate data handling throughout parts of the project, mainly when reading and writing JSON files. Secondly, the library `scikit-learn`[20] was utilized. Similarly to `TensorFlow`, it is an open-source machine learning library. However, in contrast to `TensorFlow`, it provides additional functionality `TensorFlow` itself does not offer. One of these functionalities is its `LabelEncoder`, which is used to convert string classifications (in our dataset called `tag`) into the numeric representations that machine learning models require. The other is the train-test-split, which we describe in more detail in subsection 5.3.6 Lastly, we utilized `numpy`[21] as data format passed to the ML model and in order to streamline the application of the train-test-split.

[17]https://research.google.com/teams/brain/?ref=harveynick.com
[18]https://www.tensorflow.org/api_docs/python/tf/keras/layers/ConvLSTM3D
[19]https://pandas.pydata.org/
[20]https://scikit-learn.org/
[21]https://numpy.org/

5.2 Used Datasets

As mentioned previously, the intial objective of our research was to classify source code changes across various programming languages. Consequently, we searched for repositories on GitHub whose commit messages met the specified requirements set in subsection 4.1.3. The following list presents what we found for each of the planned languages:

- C $\rightarrow$ lvgl[22]
- Rust $\rightarrow$ polars[23]
- TypeScript $\rightarrow$ node-mongodb-native[24]
- JavaScript $\rightarrow$ agoric-sdk[25]
- C# $\rightarrow$ uno.extension[26]
- Go $\rightarrow$ goreleaser[27]
- Python $\rightarrow$ commitizen[28]

For each of the datasets, 1,400 commits were extracted to reach a suitable size for the dataset.

Following the recognition that the classifier was not able to correctly predict code changes of multiple languages, we then decided to focus solely on `lvgl`. As of May 1st, 2024, `lvgl` has 10,072 commits on its main branch. Of these, we selected the most recent 5,000 for analysis. We increased the number of commits taken from `lvgl` to compensate for the overall loss of commits due to the scraping of other repositories. These 5,000 commits were subsequently reduced to 4,902 due to the absence of a patch in 98 of the commits. Furthermore, since `lvgl` has about 5.5% of its code written in a language other than C, we opted to filter out all files that did not have the file extension `.c`.

5.3 Code Snippets

The following section will present some of the most crucial code snippets that we identified throughout our implementation of the software tool.

[22]https://github.com/lvgl/lvgl
[23]https://github.com/pola-rs/polars
[24]https://github.com/mongodb/node-mongodb-native
[25]https://github.com/Agoric/agoric-sdk
[26]https://github.com/unoplatform/uno.extensions
[27]https://github.com/goreleaser/goreleaser
[28]https://github.com/commitizen-tools/commitizen

5.3.1 Extracting Classification from Commit Message

Listing 5.2: Code Snippet Extracting the Classification from the Commit Message

```
def extract_tag(message):
    pattern = re.compile(rf'^\b(?:{"|".join(TAGS)})
       \b')

    first_word_match = re.match(r'\b\w+\b', message
       )
    if first_word_match:
        first_word = first_word_match.group()

        if pattern.match(first_word):
            return first_word
        else:
            # Special cases that can be fixed
               automatically
            ...
    return None
```

This code snippet (see Listing 5.2) illustrates the basic implementation of the method `extract_tag`. This method was used in the extractor to read out the classification from the commit message. The `pattern` utilized here was inspired by a ruleset designed to enforce Conventional Commits, which we discovered in the GitHub docs (github, n.d.). Furthermore, the variable `TAGS` was defined as follows:

```
TAGS = ('build', 'chore', 'feat', 'perf', 'ci', 'fix', 'docs',
'refactor', 'style', 'test')
```

The theory behind this implementation was previously outlined in subsection 4.2.1.

5.3.2 Split Patch String

Another relevant code snippet is the division of the files' patch strings into the `added_patch` and the `deleted_patch`. The logic behind this division is depicted in Listing 5.3.

Listing 5.3: Code Snippet Dividing the Patch into the Added and Deleted Patch

```
def split_patch(patch):
    lines = patch.split('\n')

    added_patches = []
    deleted_patches = []

    prev_added_line = -2
    prev_deleted_line = -2

    for i, line in enumerate(lines):
        if line.startswith('+'):
            if prev_added_line == i-1:
                added_patches[-1].extend([line
                   [1:]])
            else:
                added_patches.append([line[1:]])
            prev_added_line = i
        elif line.startswith('-'):
            if prev_deleted_line == i-1:
                deleted_patches[-1].extend([line
                   [1:]])
            else:
                deleted_patches.append([line[1:]])
            prev_deleted_line = i

    return pd.Series({'deleted_patch':
       deleted_patches, 'added_patch':
       added_patches})
```

As depicted in Listing 5.3, the function initially `splits` the patch string into its lines of code at the newline operator ("
n"). Afterward, it iterates over each line and determines whether the line starts with either a "+" or a "-". Added lines begin with a `+`, deleted lines with a `-`. In the event that two lines of the same category are adjacent, i.e. `prev_added_line == i-1` (current line minus one), the corresponding chunk is `extend`ed by the new line of code. Lastly, by utilizing "`pandas.Series`", the split patch is returned.

5.3.3 Integration of `peewee`

As previously mentioned in the subsection on the used tools (see subsection 5.1.3), we employed the use `peewee`, in combination with a locally running PostgreSQL `Docker` container. To use a database, we first had to define the tables and their attributes.

Listing 5.4: Tables and Attributes in the PostgreSQL Datbase

```
class Commit(Model):
    url = TextField(primary_key=True)
    tag = TextField()

class File(Model):
    commit_owner = ForeignKeyField(Commit, backref=
       "files")
    id = AutoField()
    patch = TextField(null=True)
```

As illustrated in Listing 5.4, we use the `url` as the primary key for the commits. In addition to serving as the commit's primary key, the `url` also functions as a foreign key with the name `commit_owner` to the `File` entries. Since one commit can have multiple modified files, the key `id` was additionally introduced to ensure the distinguishability of the files. Joining the tables can be accomplished as depicted in Listing 5.5 like this:

Listing 5.5: Code Snippet Joining the Tables `Commit` and `File`

```
query = File.select(Commit, File).join(Commit).
  where(File.patch.is_null(False))
dataframe = pd.DataFrame(query.dicts())
```

Firstly, `peewee` is able to recognize the foreign key relation between `File` and `Commit`, and thus internally uses the join operator `file.commit_owner == commit.url`. After joining, the resulting table has `len(files)` entries, where each entry contains the combined information of the file and its related commit. In our case, each entry has the following attributes:

- commit.url
- commit.tag
- file.commit_owner
- file.id
- file.patch
- file.filename
- file.status
- file.additions
- file.deletions

The organization of the entries in this manner allows for a highly flexible approach that facilitates the straightforward application of changes. For instance, in our case, we split the patch (see subsection 5.3.2), extract the file extension and reorganize the entries to be commit-centric once more. In addition to joining the tables, we also filter out all files with no patch. As our classifier is currently configured to work with code changes and is thus reliant on a patch, we chose to remove all files that have none. This is because, for example, renaming a file does not warrant a patch. After applying the join operation, `peewee` returns a query, which we can then use to create the data frame visible in line two of the excerpt.

Table 5.1: Label Encoding vs. One-Hot Encoding

category		label encoded		one-hot encoded
`build`	$\rightarrow$	0	$\rightarrow$	[1, 0, . . . , 0]
`chore`	$\rightarrow$	1	$\rightarrow$	[0, 1, . . . , 0]
$\vdots$		$\vdots$		$\vdots$
`test`	$\rightarrow$	9	$\rightarrow$	[0, 0„ . . . , 1]

5.3.4 Label Encoding and One-Hot Encoding

As the machine learning model is unable to operate using strings, it is necessary to not only convert the patch strings into a numeric representation, but also the classifications:

Listing 5.6: Code Snippet Encoding the Classification Strings to Numeric Representations

```
    label_encoder = LabelEncoder()
    tags = label_encoder.fit_transform(df['tag'])
    tags = tf.one_hot(tags, len(set(tags)))
    tags = tags.numpy()
```

In lines one and two of the depicted code snippet (see Listing **??**), all classifications (here called '`tag`') of the dataset are passed to `scikit-learn`'s `LabelEncoder`. This enables the `LabelEncoder` to learn all possible classifications present in the dataset and return a list of n values representing the n classifications. Given that the dataset has $n = 10$ distinct classifications, the `tags` are represented by the integer values 1 to 9. Subsequently, `TensorFlow`'s `one_hot()` method is utilized to one-hot encode the tags. One-hot encoding is a method for converting categorical variables into a binary vector, where each value in the vector represents one of the categories. To illustrate, consider the ten classifications, then their label encoding and one-hot encoding would be: As illustrated above, the length of the binary vector is the length of categories (here: $n = len(['build',' chore', \ldots,' test']) = 10$). We elected not to utilize `scikit-learn`'s `OneHotEncoder` directly, as we initially sought to experiment with both label and one-hot encoding. This approach allowed for the straightforward addition or removal of the one-hot encoding. In the final step, we convert the list of encoded classifications into a `numpy` array, facilitating more efficient data handling for subsequent code.

5.3.5 Tokenize with Lexer

In Preprocessor 3 (see paragraph 4.2.2), we integrated `Pygments`, a tool used for highlighting and tokenizing code. To efficiently convert the code in our patches from strings to tokens, we wrote the following function (see **??**):

Listing 5.7: Code Snippet Tokenizing Lines of Code Using `Pygments`' `CLexer`

```
def tokenize_lexer(line_of_code):
    lexer = CLexer()
    tokens = lexer.get_tokens(line_of_code)

    token_list = []
    for token_type, token_value in tokens:
        token_type_name = string_to_tokentype(
           token_type)
        token_list.append((token_type_name,
           token_value))
    return token_list
```

The function `tokenize_lexer` is called for each line of code in our dataset. By applying `Pygments`' `CLexer().get_tokens()`, it returns an iterable of (`token_type, token_value`) pairs representing the initial line of code. Prior to returning the iterable, the method converts the `token_type` strings into actual `token types`. For instance, `"String.Double"` is converted to `"Token.Literal.String.Double"`. Although the token types are only utilized in the ML model, we opted to return both the token types and the token values for potential future research that may expand upon our work.

5.3.6 Train-Test-Split

A fundamental concept in machine learning is the train-test-split. This entails splitting the initial dataset into two parts: the training set and the test set. The training set, which typically consists of 70-90% of the entire dataset, is used to train the model. After training, the test data can then be used to evaluate the performance of the model's predictions on previously unseen data. This is a crucial assessment for the generalization of the model, as even if the model achieves a high accuracy during training, it is not guaranteed that it will perform well on data it has not encountered previously. The implementation of the train-test-split can be seen in Listing 5.8:

Listing 5.8: Code Snippet Dividing the Dataset into the Train Set and the Test Set

```
inputs_reshaped = inputs.reshape(inputs.shape[0],
   -1)
X_train, X_test, y_train, y_test = train_test_split
   (inputs_reshaped, tags, test_size=0.15)
```

In the first line of the code snippet, we reshape the dataset from five to two dimensions while retaining the first dimension. In our case, this means reshaping the dataset from (4902, 3, 3, 5, 48) to (4902, 2160). This process involves flattening the data of each commit into a single list containing all 2160 entries. This step is necessary, because `scikit-learn`'s `train_test_split()` function, which we use for the split, expects two-dimensional inputs for its execution. As inputs, we provided the list of reshaped commits, the classifications (here called `tags`) associated with the commits, and the size of the test set in percent ([0,1]).

5.3.7 Subclassed ML Model

In `TensorFlow`, there are three distinct approaches to implementing models: using the Sequential API, the Functional API, or the Subclassing API. After careful consideration, we opted for the subclassed implementation, as it offers the greatest degree of customizability. The base structure of a subclassed model is illustrated in Listing 5.9.

Listing 5.9: Code Snippet Implementing the Basics of a Subclassed Model in `TensorFlow`

```
class PatchModel(tf.keras.Model):
    def __init__(self):
        super(PatchModel, self).__init__()
        # Definition of used layers, e.g.
        self.outputs = tf.keras.layers.Dense(10)

    def call(self, inputs):
        # Application of the layers on the inputs,
           e.g.
        out = self.outputs(inputs)
        return out
```

In this example, the `PatchModel` is defined as a subclass of `TensorFlow.keras.Model`. This allows the model to overwrite the parent class's `__init__()` and `call()` functions. The method `__init__()`, which is called at the time of creation of the model, defines all variables and layers that are to be used during the training process. The `call()` function, in turn, defines how the input data is transformed by the model into the model's predictions.

5.3.8 Train and Evaluate the Model

Training and evaluating a model in `TensorFlow` is relatively straightforward. The primary syntax revolves around the functions `compile()`, `fit()`, `predict()`, and `evaluate()`.

Listing 5.10: Code Snippet Illustrating `TensorFlow`'s `compile()` and `fit()` Functions

```
model = PatchModel()

optimizer = tf.keras.optimizers.Adam(learning_rate
   =0.01)
loss = tf.keras.losses.CategoricalCrossentropy()

model.compile(optimizer=optimizer, loss=loss,
   metrics=["accuracy"])

model.fit([X_add_train, X_del_train], y_train,
   epochs=10, batch_size=batch_size,
   validation_split=0.2)
```

The code provided in Listing 5.10 is the entire code that is required for training. First, the model is defined, in this case, the subclassed `PatchModel` we created in subsection 5.3.7. Secondly, the training configuration is laid out using the `compile()` function, which includes the optimizer, loss, and metrics. For the optimizer, we utilized the `Adaptive Moment Estimation (Adam)` with a learning rate of 0.01. Since we have ten different one-hot encoded classifications, the `CategoricalCrossentropy` was chosen as the loss function. In order to assess the efficiency of the model during training, we selected the metric `accuracy` as the metric.

Once the model is configured for training, it can then be trained using the `fit()` method. We passed `fit()` the combined set of added and deleted patches ("`X_add_train`" and "`X_del_train`") in addition to the associated classifications ("`y_train`"). We also specified the number of epochs to be trained, the batch size of each sample per weight update, and the size of the validation dataset. This validation set serves a similar purpose as the test set, evaluating how well the model generalizes. However, unlike the test set, the validation set is used during training, essentially acting as a sanity check on the model during training.

Once the training of the model has finished, there are two methods for evaluating its performance. The first is to utilize the `predict()` function. As illustrated in 5.11, this method only requires the data to be predicted ("`[X_add_test, X_del_test]`"), and returns the corresponding prediction. An example of the output is presented in Table 6.1.

Listing 5.11: Code Snippet Illustration the `predict()` Method

```
model.predict([X_add_test, X_del_test])
```

The second, and arguably superior approach, involves the use of the function `evaluate()`. The code excerpt displayed in Listing 5.12 illustrates that, in addition to the dataset, the function also expects the corresponding classifications. This enables the method to calculate metrics such as the loss or the accuracy in contrast to only providing predictions.

Listing 5.12: Code Snippet Illustrating the `evaluate()` Method

```
model.evaluate([X_add_test, X_del_test], y_test)
```

6 Demonstration

Following the design and implementation of the classifications and our software tool, this section will provide a functional demonstration of our work. For this, we will outline the data sources we used for the demonstration, as well as the configuration setup for the hardware and the software. Lastly, we will present and discuss the outcomes of our demonstration in section 6.3.

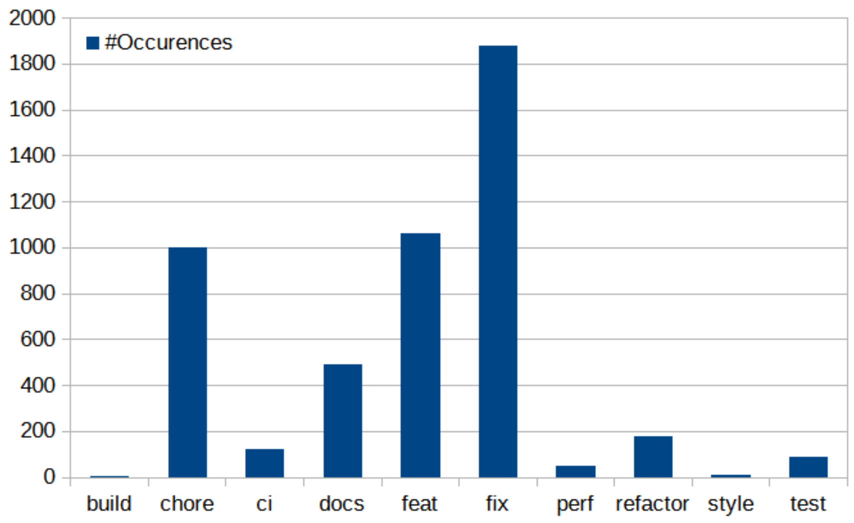
6.1 Data Sources

In this demonstration, we reuse the same dataset, comprising 5,000 commits from `lvgl`, which we previously discussed in section 5.2. The dataset was obtained using the software tool's extractor on Feb 16^{th}, 2024, meaning that the 5000 commits were created between Feb 26^{th}, 2021 and Feb 16^{th}, 2024. Given that our classifier works with patch information, we removed the commits that lacked a patch, resulting in a final set of 4,902 commits. The distribution of the classifications of the commits during this time frame is illustrated in Figure 6.1.

6.2 Configuration

In this section, we will describe the hardware and software configurations we chose for the demonstration.
In terms of hardware, we utilized an Nvidia 3050 graphics card, which we were able to control via the usage of the same `Docker` container running `gpu-jupyter` that was previously detailed in subsection 5.1.1.
As part of the software configuration, it was necessary to configure multiple parts. Firstly, we chose a train-test-split of 85%-15% to split the initial 4,902 commit dataset into training and testing data. Furthermore, the 85% training data was divided once again into 80% actual training data and 20% validation data. As a result, we get a train-validate-test-split of [0.68, 0.17, 0.15]. Secondly, we defined the following dimensions for our dataset:

Figure 6.1: Distribution of Classification Labels in `lvgl`'s 4,902 Commits (build: 6, chore: 1,002, ci: 123, docs: 495, feat: 1061, fix: 1,880, perf: 49, refactor: 182, style: 14, test: 90)



- commits/batch_size c = 48
- files f = 3
- chunks ch = 3
- lines l = 5
- words = 48

The value assigned to `c` refers to the number of data samples processed in each training iteration and is thus only partly related to the total number of commits. In addition to defining the dimensions, we also specified the size of the vocabulary, which we set to 10,000. This value is passed to the `TextVectorization` layer of the preprocessors.
Lastly, for this demonstration, we chose a combination of preprocessor 1 (see paragraph 4.2.2) and the ML model's configuration 2 (see paragraph 4.2.2).

6.3 Results

6.3.1 Functionality

Firstly, we want to demonstrate the functionality of the software tool. As illustrated in Figure 4.1, the first part of the software tool is the extractor. When examining the extractor's six steps, we can see the following:

1. Given the repository url `https://github.com/lvgl/lvgl`, the first notebook correctly writes the list of the 5,000 most recent commits to a JSON file.
2. Following the iteration of each commit in the resulting file and the extraction of each commit's `url` and `commit message`, the resulting JSON has a list of commits consisting of `url-commit message` tuples.
3. In notebook three, the extraction of the `tag` from the `commit message` is executed. All messages that follow the Conventional Commits schema and contain one of the ten classifications have their classification extracted. All remaining commits are assigned the temporary classification `None`.
4. After we manually fixed the remaining `None`-commits, the dataset includes 5,000 correctly classified commits.
5. In the fifth step, the GitHub API's response for obtaining the extended commit information is incorporated into the JSON file. In this notebook, the extractor furthermore filters out the remaining irrelevant information such that each commit has an `url`, a `tag`, and the file info remaining.
6. Finally, notebook six divides all `patch` strings into the added and deleted patches. Additionally, all files that do not contain a patch are removed such that our final JSON file includes the aforementioned 4,902 entries.

Following the extraction pipeline, the resulting JSON file is then used as input to the classifier. In the classifier, the list of all the LoCs in our dataset is passed to the `adapt()` function of the `TextVectorization` layer. To validate the correctness of the vectorized LoCs, we can check the layer's vocabulary:

```
['', '[UNK]', '0xff,', '0x00,', '=', '*', ...]
```

As can be observed, the vocabulary includes the expected words used in the patches. It is noteworthy that the entry '`[UNK]`' is a token used for all words that did not fit in the vocabulary. Afterward, the train-test-split function is applied to divide the 4,902 commits in the dataset into an 85% train and 15% test set. This can be validated by outputting the shape of the encoded classifications:

- train.shape: (4166, 10)
- test.shape: (736, 10)

Or the shape of the input dataset:

- train.shape: (4166, 3, 3, 5, 48)
- test.shape: (736, 3, 3, 5, 48)

Furthermore, since none of the shapes provided have the value `None`, it is safe to assume that the padding process done in the preprocessing worked. Upon passing the training dataset to the model's `fit()` function, the training process for the specified (ten) epochs starts and runs as expected. This is illustrated in 6.1. Additionally, it can be observed in the figure that the train-validation-split also functioned as intended.

6.3.2 Interpretation of Outputs

For the second step in the result analysis, we will examine the predictions made by the classifier given five different commits. For context, we will provide the desired one-hot encoded classification:

Table 6.1: Predicted vs. Expected Classification

ID	Expected	Predicted
1	[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]	[0.00398454 0.2505949 . . . 0.02385394]
2	[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]	[0.00398454 0.2505949 . . . 0.02385394]
3	[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]	[0.00398454 0.2505949 . . . 0.02385394]
4	[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]	[0.00398454 0.2505949 . . . 0.02385394]
5	[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]	[0.00398454 0.2505949 . . . 0.02385394]

As illustrated in Table 6.1, regardless of the input and its associated classification, the resulting prediction remains the same:

```
[0.00398454 0.2505949 0.02729493 0.08908853 0.21754928
 0.3074328 0.01806661 0.05411862 0.00801593 0.02385394]
```

When comparing the probabilities present in the predictions with the distribution of the dataset's classifications (see Figure 6.1), it becomes evident what happens. For instance, calculating the percentage share of commits classified as `feat` in the dataset, we receive $1061/4092 = 0.259$. This percentage, particularly when considering that commits are removed at random from the train set during the train-test-split, is strikingly similar to the second probability of the output. Upon considering these factors, it becomes evident that the model's prediction is not influenced by the input it receives. Instead, it merely returns the distribution of the classification, converted into their percentage share in the training set. This implies that the model did not actually learn patterns during training, but instead attempted to replicate the distribution. This lack of learning can also already be observed during the training itself, as can be seen in Listing 6.1:

Listing 6.1: Training History

```
Epoch 1/10
70/70 [==============================] - 70s 757ms/
   step - loss: 9.7805 - accuracy: 0.3830 -
   val_loss: 9.7984 - val_accuracy: 0.3921
Epoch 2/10
70/70 [==============================] - 50s 721ms/
   step - loss: 9.8972 - accuracy: 0.3860 -
   val_loss: 9.7984 - val_accuracy: 0.3921
Epoch 3/10
70/70 [==============================] - 52s 745ms/
   step - loss: 9.8972 - accuracy: 0.3860 -
   val_loss: 9.7984 - val_accuracy: 0.3921

[...]

Epoch 10/10
70/70 [==============================] - 51s 734ms/
   step - loss: 9.8972 - accuracy: 0.3860 -
   val_loss: 9.7984 - val_accuracy: 0.3921
```

After only two epochs, both the loss (distance between expected and predicted classification) and the accuracy remained unchanged. It appears that the model identified its optimal configuration at this point and does not surpass this result in any subsequent configuration.

The results illustrated above remain consistent across the six Preprocessor-ML model combinations we tested. Moreover, it was even irrelevant if we utilized a dataset including one programming language or a dataset with multiple languages. Furthermore, in order to rule out any problems with hyperparameters such as the learning rate or the batch size, we attempted manual hyperparameter optimization. However, no amount of tweaking improved the results presented above.

In conclusion, the extractor is both functional and usable. Conversely, although the classifier is functional, it is not particularly usable, as it does not meet expectations.

7 Evaluation

At this point in this thesis, we will revisit the objectives defined in chapter 3 and assess whether our solution, implementation, and demonstration align with the defined criteria.

The first objective was to ensure that our set of classifications is easily distinguishable and fully present in the majority of openly available git repositories, while also finding application in practice. In order to do this, we analyzed both research- and practice-related classification schemas, focusing on those found on GitHub. We discovered one reoccurring schema that we were able to trace back to Conventional Commits. Although rarely mentioned in scientific literature, the fact that it is a well-adapted pseudo-standard for commit messages ensures that all three subcriteria are met.

$\Longrightarrow$ Criteria 1 is fulfilled.

Secondly, we proposed that the extraction pipeline of our software tool should, given a repository, obtain a list of commits where each commit is filtered such that it only contains the information relevant to its classification. Furthermore, we aimed to include the extraction of the classification tag from the commit message in the extractor. To fulfill these criteria, we created five Jupyter Notebooks that each provide one step in the extraction pipeline. Although some classification errors (e.g. misspellings) remain to be fixed manually, the notebooks can be used seamlessly to achieve the objectives set out at the start.

$\Longrightarrow$ Criteria 2 is fulfilled.

Thirdly, we aimed to develop a classifier that could output probabilities associated with the set of classifications. While the classifier ultimately provides the same prediction regardless of the input, the goal of outputting predictions was sufficiently achieved.

$\Longrightarrow$ Criteria 3 is fulfilled.

As our fourth objective, we defined that our classifier should be usable on all types of repositories. This was demonstrated in chapter 6, where it was shown that as long as the classifier is provided the required information, it is able to predict the classifications.

$\Longrightarrow$ Criteria 4 is fulfilled.

The second-to-last criterion was to exclusively utilize data openly available on GitHub for the training of the classifier. Both prior to and after the decision to focus on a single programming language, we ensured that only repositories licensed for full open-source usage were used –in our case those were the Apache-2.0 and the MIT license.

$\Longrightarrow$ Criteria 5 is fulfilled.

The sixth and final objective we defined was to achieve an accuracy rate of at least 80 percent. This objective was not achieved. The reason for this can be attributed to the fact that the classifier did not learn any patterns from the training data, resulting in a constant prediction.

$\Longrightarrow$ Criteria 6 is not fulfilled.

8 Conclusions

In the final chapter, we will present potential explanations for why the classifier's ML model did not learn. Furthermore, we will suggest potential solutions to address the issues we identified. Lastly, we will provide an outlook for future research and suggest avenues for further improvement of the work we have done.

8.1 Problems and Fixes

Impure Data

The most probable explanation for why the classifier did not learn any patterns during training is that there were no patterns present in the data. While we ensured that each commit was classified by one of our classifications, we did not verify the correctness of this classification for every single commit. Consequently, human error at the time the commit message was composed may have played a significant role in the impurity of the dataset. For instance, consider a fix in the documentation. Conventional Commits states that this change should be classified as *docs*. However, a developer may be inclined to label the change as a *fix*, thus misclassifying the commit. Another scenario is when a developer combines multiple minor changes into a single commit. In such cases, no single classification they can choose is appropriate for the combination of changes. As a consequence, the developer should have divided those changes into two separate commits, despite the inconvenience this might cause. In order to address the problems presented by impure data, we propose one of two solutions: If the impurity of the dataset is primarily due to incorrect classifications, it should be sufficient to manually inspect all commits in order to identify and correct these misclassifications. If, however, a considerable amount of the combined commits exist, we recommend creating a new dataset that can subsequently be used for training and validation purposes.

Preprocessor Level of Abstraction

The second most probable issue that we identified is the level of abstraction introduced in the preprocessors. Despite trialing three different versions of preprocessors, none of them appeared to produce an optimal mix between keeping relevant data and removing unnecessary information. A decent balance between syntactic and semantic information might probably be provided by a preprocessor in between preprocessor 2 (see paragraph 4.2.2) and preprocessor 3 (see paragraph 4.2.2) An alternative approach would be to use both entries of the tuple returned by `Pygments`' `get_tokens()` method. This would enable the ML model to work with both the `token_type` for syntactic information and the `token_value` for semantic data. Yet another solution might be to use a pre-trained model, such as CodeBERT (Feng et al., 2020), for the vectorization of the LoCs. Similar to pre-trained models in NLP like BERT (Devlin et al., 2019) or GPT (Radford and Narasimhan, 2018), CodeBERT has previously been trained on a large corpus of relatable data. This allows these models to use transfer learning to better understand the intricacies in the code, which in turn helps them provide better encodings for the words.

Model Depth

As previously mentioned in chapter 5, we had to transition to a local development environment, limiting both GPU performance and memory when compared to `Google Colab`. As a result, we were compelled to reduce the dimensionalities of the input data per commit as well as the number of layers in the ML model. Should future research based on our approach be conducted, we recommend enhancing the computing environment to allow for more input data and layers.

8.2 Outlook and Future Research

As previously demonstrated, only the added and deleted lines of code we extracted from the patches were used to classify the commits. However, in addition to these two features, the extractor also provides more attributes such as the filename and the commit status. By using the status, for example, it would be possible to additionally include files that lack a patch but still have the required status — e.g. when the status is `renamed`. Furthermore, it would be feasible to use the `filename`, or more precisely the `file extension`, to determine which lexer to use. If a file, for instance, has the extension `.js`, the program would know to use the JavaScript lexer instead of the currently used C Lexer. Similarly, when encountering a file that ends with `.md`, it can be determined that the file is a Markdown file and can therefore be processed using NLP techniques.

8.3 Summary

To conclude this thesis, it is appropriate to summarize what we achieved in this thesis:

1. By analyzing various methods used in both academic and practical contexts to classify commits, we have identified a subset of Conventional Commits that can be widely utilized.
2. As part of our software tool, we developed an extraction pipeline using Jupyter Notebooks that extracts all information relevant for the classification of a commit and ultimately for the supervised learning of a machine learning model.
3. A first version of a classifier, including three different preprocessors and two distinct ML models, was developed using `TensorFlow`.
4. We identified potential issues that may have caused the ineffectiveness of our model and presented possible solutions for these. Additionally, we provided an outlook for future research and strategies to enhance and expand on our work.

Appendices

A Tokens in preprocessor 2

Listing 1: List of Tokens Utilized in Preprocessor 2

```
tokens = {
    '::': 'SCOPE',
    '\\+\\+': 'INCREMENT',
    '--': 'DECREMENT',
    '\\+=': 'PLUS_EQ',
    '\\+': 'PLUS',
    '-=': 'MINUS_EQ',
    '-': 'MINUS',
    '\\(': 'ROUND_OPEN',
    '\\)': 'ROUND_CLOSE',
    '\\[': 'SQUARE_OPEN',
    '\\]': 'SQUARE_CLOSE',
    '\\{': 'CURLY_OPEN',
    '\\}': 'CURLY_CLOSE',
    '\\.': 'DOT',
    '->': 'POINTER_SELECTION',
    '!=': 'NOT_EQ',
    '!': 'LOGIC_NOT',
    '~=': 'BIT_NOT_EQ',
    '~': 'BIT_NOT',
    '\\*=': 'STAR_EQ',
    '\\*': 'STAR',
    '/=': 'DIV_EQ',
    '/': 'DIV',
    '%=': 'MOD_EQ',
    '%': 'MOD',
    '<<=': 'LEFT_SHIFT_EQ',
    '<<': 'LEFT_SHIFT',
    '>>=': 'RIGHT_SHIFT_EQ',
    '>>': 'RIGHT_SHIFT',
    '<=': 'LESS_EQ',
    '<': 'LESS',
    '>=': 'GREATER_EQ',
    '>': 'GREATER',
    '==': 'EQUAL',
    '\\^=': 'XOR_EQ',
    '\\^': 'XOR',
    '&&': 'AND',
    '&=': 'BIT_AND_EQ',
    '&': 'BIT_AND',
    '\\|\\|': 'OR',
    '\\|=': 'BIT_OR_EQ',
    '\\|': 'BIT_OR',
    '=': 'ASSIGN',
        ',': 'COMMA',
        ';': 'SEMICOLON'
    }
```

References

Angular Team. (n.d.). *Angular convention* [Accessed: 2024-04-04]. https://github.com/angular/angular/blob/22b96b9/CONTRIBUTING.md#-commit-message-guidelines

Casalnuovo, C., Suchak, Y., Ray, B., & Rubio-González, C. (2017). Gitcproc: A tool for processing and classifying github commits. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 396–399. https://doi.org/10.1145/3092703.3098230

ConventionalCommits.org. (n.d.). *Conventional commits* [Accessed: 2024-04-03]. https://www.conventionalcommits.org/en/v1.0.0/

De Wilde, M. (2019). Automatic git commit generation and classification for html based projects.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Docker Inc. (n.d.). *Docker overview* [Accessed: 2024-04-14]. https://docs.docker.com/get-started/overview/

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020, November). CodeBERT: A pre-trained model for programming and natural languages. In T. Cohn, Y. He & Y. Liu (Eds.), *Findings of the association for computational linguistics: Emnlp 2020* (pp. 1536–1547). Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Gaviar, A. (2019). *Github's top 100 most valuable repositories out of 96 million* [Accessed: 2024-04-03]. https://hackernoon.com/githubs-top-100-most-valuable-repositories-out-of-96-million-bb48caa9eb0b

github. (n.d.). *Rulesets-commit-regex* [Accessed: 2024-04-15]. https://github.com/github/docs/blob/d8df1ab88f1f61cbb90d55715be2ad43b5892cbd/data/reusables/repositories/rulesets-commit-regex.md

Google Brain Team. (n.d.-a). *Introduction to tensorflow* [Accessed: 2024-04-30]. https://www.tensorflow.org/learn

Google Brain Team. (n.d.-b). *Tf.keras.layers.textvectorization* [Accessed: 2024-04-11]. https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization

Hattori, L. P., & Lanza, M. (2008). On the nature of commits. *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, 63–71. https://doi.org/10.1109/ASEW.2008.4686322

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Q.*, *28*(1), 75–105.

Hindle, A., German, D. M., Godfrey, M. W., & Holt, R. C. (2009). Automatic classication of large changes into maintenance categories. *2009 IEEE 17th International Conference on Program Comprehension*, 30–39. https://doi.org/10.1109/ICPC.2009.5090025

Hoang, T., Lawall, J., Tian, Y., Oentaryo, R. J., & Lo, D. (2021). Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel. *IEEE Transactions on Software Engineering*, *47*(11), 2471–2486. https://doi.org/10.1109/tse.2019.2952614

iot-salzburg. (n.d.). *Gpu-jupyter* [Accessed: 2024-04-15]. https://github.com/iot-salzburg/gpu-jupyter

Jupyter Team. (n.d.). *Jupyter notebook documentation* [Accessed: 2024-04-14]. https://jupyter-notebook.readthedocs.io/en/latest/

Leifer, C. (n.d.). *Peewee* [Accessed: 2024-04-15]. https://docs.peewee-orm.com/en/latest/

Levin, S., & Yehudai, A. (2017). Boosting automatic commit classification into maintenance activities by utilizing source code changes, 97–106. https://doi.org/10.1145/3127005.3127016

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, . . . Xiaoqiang Zheng. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems [Software available from tensorflow.org]. https://www.tensorflow.org/

Meng, N., Jiang, Z., & Zhong, H. (2021). Classifying code commits with convolutional neural networks. *2021 International Joint Conference on Neural Networks (IJCNN)*, 1–8. https://doi.org/10.1109/IJCNN52387.2021.9533534

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Radford, A., & Narasimhan, K. (2018). Improving language understanding by generative pre-training. https://api.semanticscholar.org/CorpusID:49313245

Sarwar, M. U., Zafar, S., Mkaouer, M. W., Walia, G. S., & Malik, M. Z. (2020). Multi-label classification of commit messages using transfer learning. *2020 IEEE International Symposium on Software Reliability Engineering Work-*

shops (ISSREW), 37–42. https://doi.org/10.1109/ISSREW51248.2020.00034

Shi, X., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-k., & Woo, W.-c. (2015). Convolutional lstm network: A machine learning approach for precipitation nowcasting.

Swanson, E. B. (1976). The dimensions of maintenance. *Proceedings of the 2nd international conference on Software engineering*, 492–497.

T., T. (2023). *15 most popular github repositories every developer should know* [Accessed: 2024-04-30]. https://www.hostinger.com/tutorials/most-popular-github-repos

Tong, J., Wang, Z., & Rui, X. (2023, August). *Boosting commit classification with contrastive learning*. https://doi.org/10.2139/ssrn.4740998

Westlake, R. (2017). *Write categorized git commit messages* [Accessed: 20124-04-03]. https://medium.com/@rcwestlake/write-categorized-git-commit-messages-c9f953ea6040

Ziegelmayer, F. (n.d.). *Git commit msg* [Accessed: 2024-04-04]. https://karma-runner.github.io/0.10/dev/git-commit-msg.html