Detecting PII in Git commits

N. van der Plas

Master of Science Thesis

TU Delft Delft University of Technology

Delft Department of Software Technology

Detecting PII in Git commits

Master of Science Thesis

For the degree of Master of Science in Computer Science at Delft University of Technology

N. van der Plas

June 28, 2022

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS) · Delft University of Technology

The work in this thesis was supported by ING. Their cooperation is hereby gratefully acknowledged.

Copyright © Delft Department of Software Technology (ST)
All rights reserved.

Abstract

With the advancement of technology, organizations are experiencing more trouble with keeping their data private with it often leaked to the public via their code-repositories or databases. There are methods to counter the leakage of data while pushing code to a repository however, these are heavily reliant on regular expressions. Personal names, locations and other Personally Identifiable Information (PII) do not follow a reoccurring pattern and can thus only be prevented by manual code reviews, which are also prone to errors. A tool to detect these PII should be designed as an initial measure to counteract the leakage. In this paper, we propose a heavily modifiable tool in which we combine the strength of regular expressions with a state-of-the-art machine learning model to detect a variety of important PII within the code changes of Python software projects. We use CodeBERT, a RoBERTa-like Transformer model, as our PII recognizer. This recognizer is fine-tuned using the Scikit-learn library of which we injected the git commits with fake sensitive data. To test and improve the quality of the model and the entire tool, we design an experimental methodology to find the optimal value for the hyper parameters of the model, compare it against another Transformer model and run the fine-tuned model against several other code-bases with different programming languages. The outcome of these experiments benefit the quality of the model in a positive way and allows us to design a robust tool with a well-performing machine learning model to detect a variety of entities. This tool can be personalized to any business and mitigate a significant part of the potential data leaks.

Table of Contents

Preface xi

1 Introduction 1

1-1 What is PII? 1
1-2 How does PII get leaked in Git commits? 2
1-3 Existing tools to combat data leakage within Git 3
1-4 Research effort / Aim of the thesis 3
1-5 Outline of the thesis 4

2 Background 5

2-1 PII for ING 5
2-2 Transformers 5
2-2-1 Huggingface library 6
2-2-2 Named Entity Recognition (NER) 6
2-2-3 CodeBERT 6
2-3 Related work 7
2-3-1 Publicly available tools 7
2-3-2 Academic work 8
2-3-3 Previous efforts ING 10

3 Relevance for ING 13

3-1 Goal 14
3-2 Technical overview 14

4 Methods **15**
4-1 Data 15
4-1-1 Clean data set 15
4-1-2 Pre-processing 15
4-2 Formatting 16
4-3 Injecting sensitve data 17
4-3-1 Sources for sensitive data 17
4-3-2 PII in the data set 18
4-4 Model training 18
4-4-1 Ktrain 18
4-5 Validation 19
4-6 Regex & Checksum 19
4-6-1 Checksums 20
4-6-2 Entities and their checksum 21
4-7 Full framework 22

5 Results **23**
5-1 Hyper parameters 23
5-1-1 Learning rate 23
5-1-2 Batch size 25
5-1-3 Number of epochs 25
5-2 RoBERTa vs. CodeBERT 26
5-3 Validation on other data sets 27
5-3-1 Python language 28
5-3-2 JavaScript language 29
5-3-3 C++ language 29
5-4 Scraping git commits to find PII 30

6 Discussion **31**

7 Conclusion **33**

A Appendices **35**
A-1 Specifications 35

List of Figures

1-1 The flow diagram of Git secret scanning as shown in [7] 3

2-1 The high-level design of the Presidio Analyze 8
2-2 The distribution of max. line length in clean data and data with leaks injected . 12

3-1 The flow diagram of the Data leakage prevention system 14

4-1 Framework of the proposed tool 22

5-1 Learning rate plot 24

List of Tables

2-1 The sensitive PII as identified by ING stake-holders 6
2-2 The results of initial baseline model 11

4-1 The number of occurences per entity 18
4-2 Regular expression matched entities and their respective validation method . . . 21

5-1 F_1 scores of the entities on different learning rates 24
5-2 Accuracy and time to completion per epoch 26
5-3 RoBERTa(left) and CodeBERT(right) scores 26
5-4 Results of prediction on the Flask repository with injected entities 28
5-5 Results of prediction on the React Date Picker repository with injected entities . 29
5-6 Results of prediction on the ImGui repository with injected entities 30
5-7 Entities found in psf/requests 30

Listings

4.1 Git diff body snippet from commit '8d3b424' 16
4.2 Sensitive data injected code 16
4.3 Probability of randomness calculation 20
5.1 Code snippet in which entities are embedded 27
6.1 Entities recognized with cased examples 31

Preface

Dear reader,

With this thesis an end is coming to my years of studying. I started off at Leiden University where I did my Bachelor and then at the Technical University of Delft. In these years I've met many people who gave me great advice along the way and who helped me get to this point and I am grateful for every one of them. I chose for the TU as to gain a better understanding about the topics I am interested in and for the freedom they give you as a person to find out what you like. Thinking back, I made the right choice. During my thesis I've met great people, within the TU and outside, who helped me write and supported me when things didn't go as planned during the pandemic. That is why I would like to thank Assistant Professor Luís Cruz for his support and guidance all throughout the process of my thesis. During the weekly meetings, he would steer me in the right direction and give feedback on what I had done.

Another important person in this process is PhD. Luiz O. V. B. Oliveira who was my buddy and supervisor within ING. Luiz was always happy and interested in what I was doing and helped me when I was stuck. I want to thank Luiz for the great moments we shared, the proof-reading and the time he took for me when he thought I needed it.

Finally, I would like to thank my girlfriend, friends and family for sharing their thoughts and helping me in any way during this process.

I hope that this thesis is as interesting to read as it was for me to write and carry out. Enjoy.

Delft, University of Technology
June 28, 2022

N. van der Plas

“The road goes ever on and on
Down from the door where it began
...
Still round the corner there may wait
A new road or a secret gate,
And though we pass them by today,
Tomorrow we may come this way
And take the hidden paths that run
Towards the Moon or to the Sun.”
— *J.R.R. Tolkien*

Chapter 1

Introduction

Many established businesses are struggling to keep their Personally Identifiable Information (PII) in-house [1], [2] and, with the fast increase in processing power to search through an growing amount of public repositories, the risk for solo developers as well as entire organisations to leak their PII is growing. As it currently stands, there exist tools that can help prevent leaking PII in Git commits. However, most of these work in a specific context and through either pattern matching or calculating Shannon's entropy [3]. In the academic world, this topic has also gained more traction and quite a lot of work with the use of machine learning or deep learning approaches has been published. There is not enough research or experiments done to evaluate the effectiveness of the proposed techniques and thus their effectiveness in the industries though. There is an ever-growing need for better, well-evaluated and more accurate detection systems in free-text as well as in code, particularly in large businesses.

For ING, which is a global financial institution, a reasonably fast, reliable and good solution to the problem of PII detection and leakage prevention is needed. With their large customer base and the amount of people that rely not only on their services but also their integrity of safe-keeping their assets and information, leakage of PII risks hurting the trustworthiness of ING. This could lead to fewer investments from individuals and larger institutions resulting in lower revenue and profits or investors pulling away from investing in the ING group at all. Leakage of secret or API keys could also disturb other services ING provides. Depending on the severity of the leak it could lead from downtime of their payment services to attackers gaining access to ING data.

1-1 What is PII?

PII stands for Personally Identifiable Information and is defined by the National Institute of Standards and Technology (NIST) as follows [4]:

“PII is any information about an individual maintained by an agency, including (1) any information that can be used to distinguish or trace an individual‘s identity, such as name, social security number, date and place of birth, mother‘s maiden name, or biometric records; and (2) any other information that is linked or linkable to an individual, such as medical, educational, financial, and employment information.”

Furthermore, API keys or IDs are also considered Personally Identifiable Information or *sensitive* for many organizations, including ING.

1-2 How does PII get leaked in Git commits?

Software is often designed to deal with PII albeit never fool-proof, so developers need to be extremely careful to prevent leaking PII into any codebase. Not only does this result into stress for the developers but some mistakes will always be made at some point as well and a single mistake can lead to massive losses especially in highly regulated organizations.

There are multiple ways that PII can be leaked and exposed to actors that want to do harm:

- By straightforwardly pushing code consisting of PII to Git repositories. This can happen in instances where a developer has a comment in which he would store an actual email as an example input for a function or when an API key is declared at the top of a code file and forgotten about. If it were to be deleted in a future commit it could still be found by analyzing the commit history. In the commit history all the diffs are stored; these diffs contain the changes made in a file for all changed files in a repository. In a scenario where a developer pushed a bank account number to a repository, he might think that when he deletes the account number in the next commit and push to the repository again his mistake would be fixed. However, the account number is still stored in the diff of the two commits, the one where he adds it and the one where he deletes it. An attacker that is out for these sort of data leaks could use a pattern recognizer to scan through all the commits in the commit history, and the account number would be compromised.

- By pushing files that should have been in a blacklist file. This is a file that keeps track of documents that it should not include when committing to a repository. Using a blacklist file, one can blacklist files with a certain extension, name or whole directories. An example of such a file in practice is ‘.gitignore’ in Git. When working with a remote server to which access is desired, developers or engineers often choose the SSH protocol in which they generate a pair of public and private keys, which are files that encompass the keys needed to access the remote server. When these files are not included in the ‘.gitignore’ they can be pushed by mistake to the repository and are consequently available to anyone who has access. This can have negative consequences, mainly in public repositories, as everyone can access and dig through them.

1-3 Existing tools to combat data leakage within Git

While GitHub is one of the world's most popular and trusted site for storing code and other related files, thousand of new unique secrets and other PII are leaked every day [5]. There are existing tools Git provides that try to prevent or get rid of PII in repositories. Git provides a function called 'git filter-repo' that purges a file from a repository's history, completely rewriting the Git history without any trace or mention of the file purged, by changing the SHAs for the existing commits that are altered and the dependent commits [6].

GitHub also has a service in place called 'secret scanning', which is automatically enabled on public repositories [7]. Secret scanning will scan the entire Git history on all the existing branches for any secrets such as API or access keys. GitHub then verifies these keys by their service providers, if the partners have partnered up with GitHub to do so. If the key is verified, the provider is notified and could then choose whether to revoke the secret, issue a new secret or notify the one who generated it.

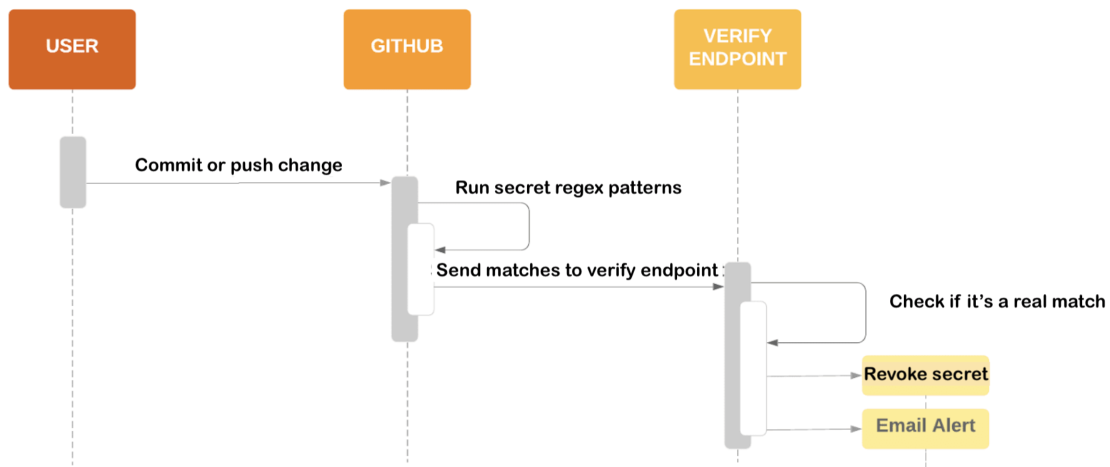

Figure 1-1: The flow diagram of Git secret scanning as shown in [7]

In Figure 1-1, the flow diagram of Git's secret scanning is displayed; whenever a user commits or pushes to a branch, GitHub will run a wide variety of secret regex patterns as supplied by the companies that have partnered up with GitHub. The secrets found are then verified against their respective endpoints at the partners to check if it is a valid secret. The partner can then choose to notify the owner of the repository where the push occurred or to revoke the secret.

1-4 Research effort / Aim of the thesis

The aim of this thesis is to create a tool that detects PII in committed Python code. A tool consisting of a combination of existing techniques and, next to pattern and entropy matching, our tool will be enhanced with a BERT model [8] specifically trained on Named Entity Recognition of entities important to ING in code. This tool needs to be fast, have

high recall (so that almost all relevant results are detected) and high precision as we don't want to disturb the developers with false positives. While creating this tool we will answer multiple questions that are important in the process:

1. Can we use synthetic data to train models that identify PII in code commits?
2. Can a model fine-tuned on Python code detect PII reliably in different programming languages?
3. How does our model perform on existing open source Python projects?

1-5 Outline of the thesis

The remainder of this thesis consists of six chapters. In Chapter 2 the related work will be discussed after which in Chapter 3 the relevance for ING is explained more thoroughly. Next, a tool will be proposed and every element of it explained in Chapter 4 followed by an evaluation of this proposed tool in Chapter 5. We discuss and conclude the thesis research by reflecting on our efforts in Chapter 6 and 7.

Chapter 2

Background

2-1 PII for ING

PII for ING is any data that is not code that can be traced back to individual clients. In the recognition of PII we take note of the difference between 'linked' and 'linkable' PII. NIST defines linked and linkable information as follows [4]:

Linked information: "information about or related to an individual that is logically associated with other information about the individual"

Linkable information: "information about or related to an individual for which there is a possibility of logical association with other information about the individual"

Linkable information can be quite tricky as on its own it may not be able to identify a person, but when combined with another piece of information it could identify, trace or locate a person. With this in mind, we decided to treat the linkable information as sensitive information and decided to incorporate it in our model.

Table 2-1 contains a large part of the sensitive PII that should be prevented from leaking as identified by the stakeholders at ING. We will discuss later in Section 4-6 our approach on how to detect and validate these entities.

2-2 Transformers

In 2017, a novel and promising approach to natural language processing was published which had a significant impact on the NLP (Natural Language Processing) landscape. Vaswani et al. [9] proposed a new method to address the complexity of the natural language, specifically on the large NLP tasks such as question-answering and text classification. This new model introduced an 'attention' (hence the name of the paper) mechanism which considers the relationship between all the words in the sentence. The Transformer was able to drastically improve on the NLP tasks by a significant margin and with a fraction of the computational

Linked	
Attribute name	Personal information
IBAN	Yes
Email	Yes
Transactions	Yes
Corporate keys	Yes
Full names	Yes
Website	Yes
API keys	No
Passwords	No
BSN/SSN	Yes

Linkable	
Attribute name	Personal information
Country	No
Town	No
First name	Yes
Last name	Yes

Table 2-1: The sensitive PII as identified by ING stake-holders

cost and as a result considered state of the art. The introduction of the Transformer kick-started extensive research in the NLP field as well as improvements that could be made with the model. In recent times, many now considered state-of-the-art models incorporating the Transformer in their architectures have made headway. Some of the more popular models are Google's BERT proposed by Devlin et al. [8] and OpenAI's GPT models [10].

2-2-1 Huggingface library

The Huggingface Transformers library[1] is a popular library for state-of-the-art NLP which provides a variety of (community-uploaded) pre-trained language models which are based on the Transformer architecture as well as a lot of examples, tutorials and scripts to fine-tune your own model or to create a downstream task like Named Entity Recognition. The project has a large community which is very active and help to develop the library.

2-2-2 Named Entity Recognition (NER)

According to Li et al. [11] 'Named Entity Recognition is the task to identify mentions of rigid designators from text belonging to predefined semantic types such as person, location, organization etc'. A Transformer can be fine-tuned through training to identify such rigid designators in any context by supplying it with a large quantity of samples.

2-2-3 CodeBERT

While researching the topic of detecting entities in code, we came across an academic paper by Karmakar et al. [12] which found that BERT models perform surprisingly well on some code tasks, indicating that BERT as well as similar models like RoBERTa are able to gain knowledge of programming languages while pre-trained on free text. While BERT already performed suprisingly well and calls for further investigation in code tasks according to the paper, CodeBERT, a RoBERTa-like model pre-trained on source code and natural language

[1] https://huggingface.co/

documentation of six different languages, performed even better suggesting that pre-training on code has impact on the results after fine-tuning on task specific data. Our model's task is to classify tokens in code and with the information and results from Karmakar's research, our intent is to investigate further the performance of both RoBERTa and CodeBERT.

2-3 Related work

There exist many projects that attempt to detect sensitive data in Git repositories, often also digging through the commit history. These tools all have their own approach, boiling down to three basic idea's:

- Regular expression pattern recognition
- Entropy checks
- Machine learning algorithms

In this section, the most influential and commonly used tools will be discussed, as well as academic papers on the topic.

2-3-1 Publicly available tools

TruffleHog

TruffleHog[2] is a Python tool that aids developers to avoid accidentally leaking secret keys on GitHub by searching through their repositories (branches and commit history included) for both high-entropy strings and patterns that match a standard. For entropy checks, truffleHog will evaluate the Shannon entropy for both the base64 and hexadecimal character sets for every blob of text greater than 20 characters comprised of those character sets in each diff. If a high entropy string with more than 20 characters is detected at any point, it will print to the screen. As truffleHog works with regular expression patterns and entropy matching only, it is significantly faster than solutions with machine learning models.

Presidio

Presidio[3] is a PII analyzer and anonymization tool created by Microsoft and can help identify and anonymize sensitive/PII data in structured/unstructured text leveraging Named Entity Recognition, regular expressions, context, rule-based logic and checksum. The analyzer tool would first use pattern recognition to detect all supported patterns in the text. After that, it would use machine learning leveraging natural language to detect entities. To reduce false positives, the tool will then try to use checksum to validate patterns if they can be validated and look for context words around detected patterns or entities to increase confidence scores.

[2] https://github.com/trufflesecurity/trufflehog
[3] https://github.com/microsoft/presidio

The Presidio architecture is composed of two main modules for anonymizing PII in text: Presidio Anonymizer and Presidio Analyzer. Presidio Analyzer runs a set of different PII recognizers, each one in charge of detecting one or more PII entities using different mechanisms (see Figure 2-1). It can also be easily extended to support additional entities. The package includes a wide range of regular expression based pattern recognizers to detect IBAN, tax file numbers, driving license numbers, individual taxpayer-identification numbers, passport numbers, IP addresses etc. It uses the Spacy NLP package for NER.

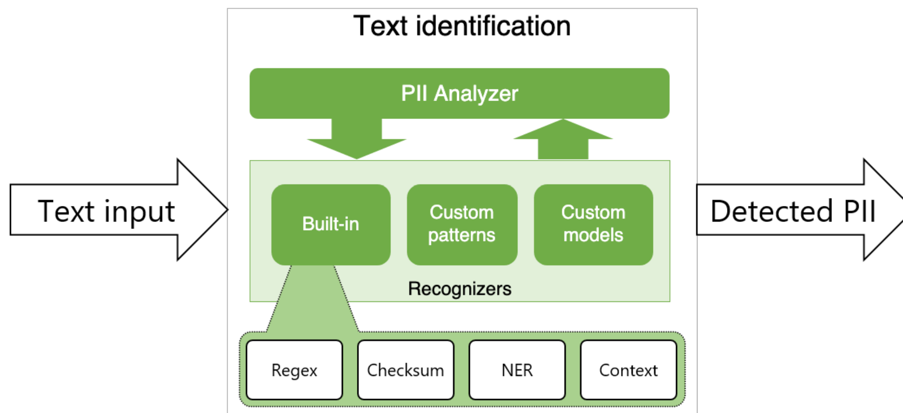

Figure 2-1: The high-level design of the Presidio Analyze

GitLeaks

Gitleaks [13] is a Static Application Security Testing (SAST) tool for detecting secrets such as passwords or API keys in git repositories. It is a very similar tool to truffleHog.

PrivAPI

PrivAPI[4] is a Python package that detects personal data within REST API communication using Deep Neural Networks (DNN). In order to generate a training data set containing positive and negative cases, it synthesizes request payloads imputed with personally identifiable information (PII). The samples containing PII were then over-sampled to balance the training data set. The model is trained to learn sequences containing both the label for each personal information and the format of the respective values. PrivAPI makes use of a Long-Short Term Memory (LSTM) neural network for sensitive data detection. The model is currently experimental and does not include adequate performance evaluations.

2-3-2 Academic work

The related academic work on detecting or predicting personally identifiable information in code is minimal. There exists some academic work on the matter in other contexts such as

[4] https://github.com/Veridax/privapi

free-text or on the detection of other information closely related to PII such as keys. While their context is not the same as ours, their approaches are still interesting and could be of use for our application. In this section, we will discuss their approaches.

Prediction of PII in emails

Geng et al. [14] present two strategies to predict whether there is PII in emails. It is worth noting that they do not try to identify the PII in the email but instead predict whether the email contains PII or not. The first method is using association rule mining to predict private information according to other PII identified. Association rule mining is a data mining approach that discovers items that frequently co-occur within a data set. With an already annotated data set for four PII (email addresses, phone numbers, addresses and money), the authors were able to mine association rules. E.g., the rule

```
EMAIL=true ADDRESS=true ==> PHONE=true conf:(84%)
```

corresponds to stating that if the email contains an email address and an address, it also contains a phone number according to the data set, with confidence of 84%.

For the second method, the authors used classification models to predict the PII according to the content of the emails. After pre-processing an email by removing stop words and stemming all other words, the authors select a subset of the remaining words and use them as features for the classification model.

Detecting keys in source code repositories

Sinha et al. [15] outline methods for detecting API keys, evaluate their effectiveness and enumerate existing mechanisms that could be used to mitigate possible leaks. They conclude by sketching a possible solution that is a combination of existing techniques. The current available methods that are acknowledged by the authors can be categorized in four different categories:

1. Sample selection using keyword search

   The method looks for keywords or filenames that may indicate the presence of keys. For example, a developer might look for the string 'BasicAWSCredentials' which takes a clientID and key as input to find a possible secret key about to be leaked nearby.

2. Pattern-based search

   Many keys and other PII follow a certain pattern. An email contains an '@' followed by a domain and an AWS clientID start with 'AKIA'. An application of pattern-based search could be to use regular expression matching over every file of a repository for these credentials or PII. This method is very fast but is prone to preduce a lot of false positives.

3. Heuristics-driven Filtering

   This is to filter out false positives. An application of this method could be to filter out instances where a match for a clientID and a secret key don't appear within 5 lines of

eachother. Calculating the entropy of a pattern-matched string is also an example of this filtering. This way false positives, where the key was a placeholder put in place by someone, could be filtered out as the entropy would be rather low. Auto-generated keys are random and will thus have a higher entropy.

4. Source-based Program Slicing

   Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow [16] and, in this context, is used to find corresponding values to the parameters of a specified function call. When using this method on the constructor call of 'BasicAWSCredentials', the algorithm would run through the entire project and find the values passed as the parameters to this constructor call.

The authors move on to tools that could be useful to protect developers and organisations against accidental key leaks and mention some prevention techniques and remedying options. They conclude by proposing a simple solution combining the methods in the four categories and presents it as a pre-commit hook. Before a developer commits code, the tool checks the edits and if a leak is detected, it returns a warning to the user. It is worth noting that this proposal focuses on keys and secrets and not necessarily PII. However, as our intention is to detect keys and secrets as well, the methods categorized could still be useful and integrated in our proposed tool.

2-3-3 Previous efforts ING

The Data Leakage Prevention System project dates back to 2020 when two engineers of ING started and maintained the project and successfully deployed a classifier model in 2021. The intention of the initial model was to cover leaks involving IBAN, customer names, email addresses, API keys and passwords in Python files as a baseline.

Training

Training a model for the system proved to be quite difficult as there was no labeled data within ING in regards to the sensitive data. To overcome this, the team took Python files from the Scikit-learn library as clean data and sampled sensitive data which was injected in the clean data. The files were used as training data and used to train a Random Forest Classifier. An example of a sample of a sensitive data entry is as follows with *'s on places where actual sensitive data is:

```
2021-09-02,***************, nan, nan, nan, ******, ****, ...
```
(Actual samples contain even more columns)

This line would be injected as a whole somewhere in a Python script contained in Scikit-learn library. The goal of the model was to identify whether a file contained a data leak or not, a binary classification.

Features

The team of engineers determined six data features which the Random Forest Classifier is trained on.

Metric	Value
Accuracy	0.993
F1	0.963
Precision	0.93
Recall	1.00

Table 2-2: The results of initial baseline model

1. Entropy
2. Avg. line length
3. Max. line length
4. Min. line length
5. No. of lines
6. No. of appearances of some keywords ('and', 'def', 'for', ...)

Results

The Random Forest Classifier performed very well with these features as can be seen in table 2-2.

However, when taking note of the example as given above the attentive reader might notice a problem with how this model is trained. One of the features used to train the model is the maximum length of a line and, while Python code line lengths usually are not **that** long, the injected sensitive data lines are. Taking into consideration the distribution of the maximum line length as shown in figure 2-2, it becomes clear that the Random Forest Classifier converged to a classification on the maximum line length instead of a combination of the aforementioned features. The results of this model were thus nugatory.

Future

The next steps in this project should be to train a new model and validate it properly so that it performs as it should. The prediction scope also needs to be improved to predict the data leaks per line instead of per push. Another improvement would be to predict the type of data leak e.g. an IBAN or website.

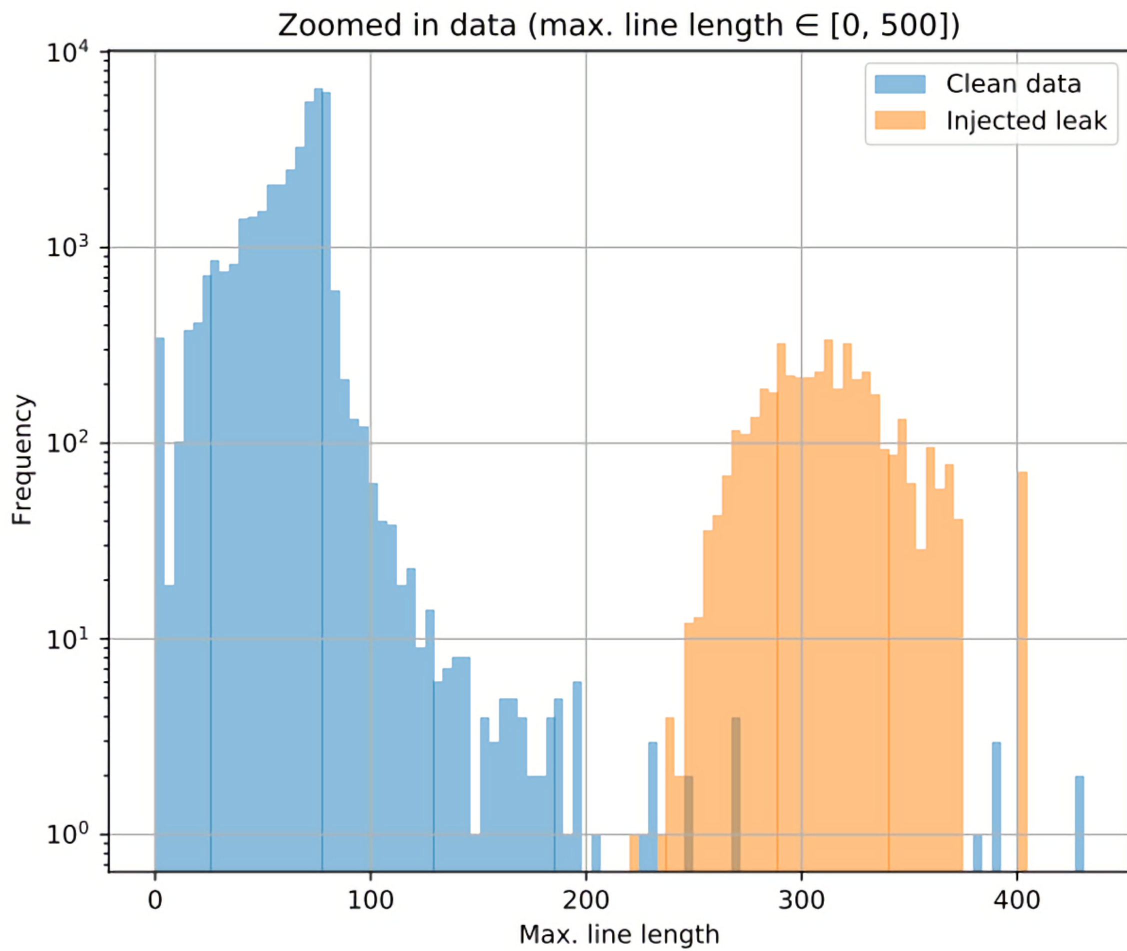

Figure 2-2: The distribution of max. line length in clean data and data with leaks injected

Chapter 3

Relevance for ING

ING is a global financial institution with more than 57.000 employees serving 38 million customers, corporate clients and other large financial institutions world-wide [17]. With services in savings, payments, investments and wholesale banking, ING bares a great responsibility to their clients to keep their data and financial details safe. Leaking customer data or financial details accidentally could be detrimental for a financial institution as they rely on their customers which, after such scandal, might look somewhere else to store their savings or investments.

The data leakage prevention machine learning model is part of a bigger system set up by ING engineers. The bigger system holds the entire pipeline from the commit to scanning to pushing. In figure 3-2 a technical overview of this system is given.

This system is integrated within the Data Analytics Platform (DAP), a platform created by ING engineers. The Data Leakage Prevention System is only useful for this platform and its data scientists and data engineers as here personal information of customers and other organisations can directly be accessed by data scientists and engineers. DAP is a highly-secure, self-serviced platform which aims to address all analytic needs. As engineers work in DAP and push their code to repositories in Azure DevOps, PII could be pushed with the code by accident which should be avoided since then it would be in the repositories of Microsoft which could access it themselves if they want to or leak it in a data breach. The model proposed could also be useful to other teams outside of the data science spectrum as these engineers do still work with API keys and secret that might accidentally be leaked.

That is why ING advocates for a 'Data Leakage Prevention System' (DLPS) to be integrated in the workflow of their engineers.

An initial PII detection model for the service was developed, using the entropy to classify content as explained in Section 2-3-3. However, this approach is very simplistic and did not work as intended, which lead to removal from the current setup.

3-1 Goal

The goal, as established by ING stakeholders, is to:

"leverage DLPS to keep DAP data secure while having an open connection from DAP to Azure DevOps and at the same time making this security invisible to users and aim for as little as possible extra user actions while pushing code to Azure".

3-2 Technical overview

Figure 3-1 shows the technical overview of the entire Data Leakage Prevention System. This thesis focuses solely on the DLPS classification module as identified in the image, the rest of this overview was already in place to suit the needs for the previous system and is not important for this research.

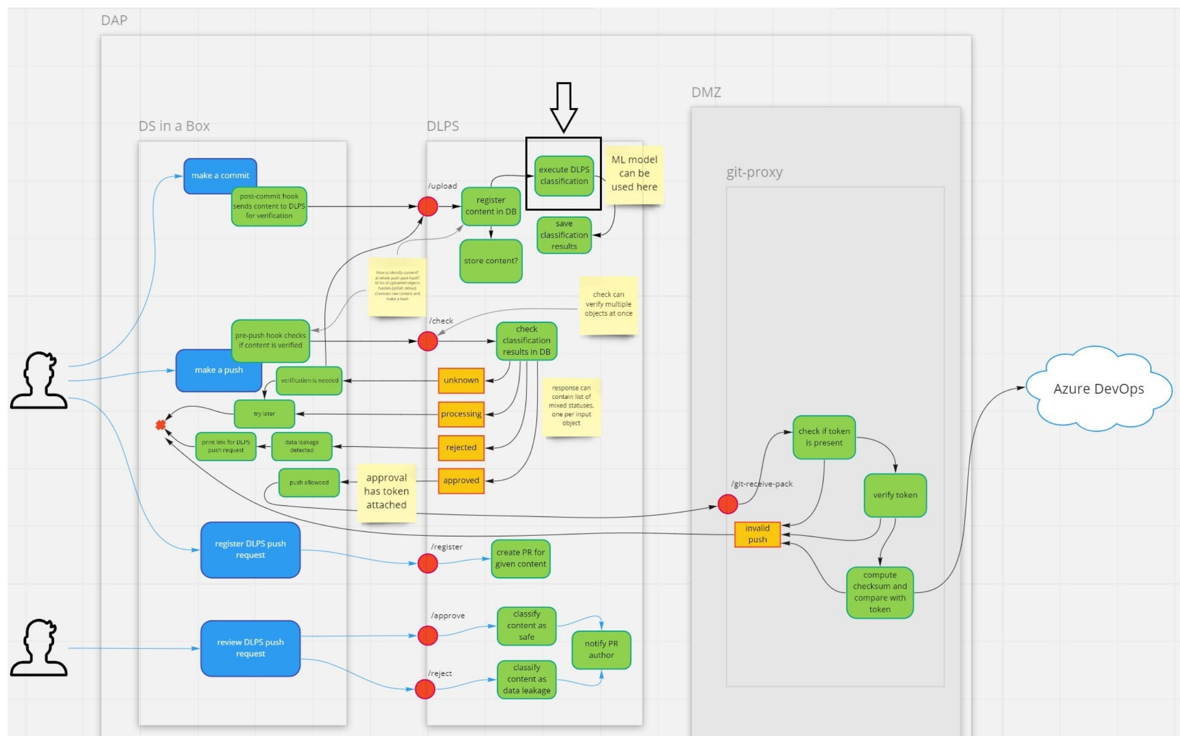

Figure 3-1: The flow diagram of the Data leakage prevention system

Chapter 4

Methods

To design a tool which is able to detect the PII relevant to ING consisting of keys, personal data and bank accounts, multiple key elements should be in place. The most important would be our Named Entity Recognizer model which needs to be trained on data that contains sensitive data. In this section we introduce the data adopted, the model and the approach used for training.

4-1 Data

Training a model to recognize the relevant entities in code requires a large labeled data set. However, although the code base owned by ING is big enough for training new models, it would involve a lot of manual work to label the data. Thus, in our approach, we take a clean data set (clean in this context means that we assume it to contain no PII in the original Git commits), pre-process this data and inject sensitive data in it.

4-1-1 Clean data set

As the task of the model is to identify sensitive data in code, we chose the git diffs of the Scikit-learn repository as clean data [18]. While the repository might contain some sensitive data in its commits, this number is negligible in comparison to the number of injected sensitive data. In total, 20.000 commits were scraped dating from March 2nd, 2021 (commit '28ee486') back to the November 16th, 2016 (commit '07b39af'). To scrape the commits we used PyDriller [19].

4-1-2 Pre-processing

With the intention of detecting whether or not a developer is pushing sensitive data in Python files to Azure DevOps, we pre-process the data set we got from Scikit-learn by only selecting Python files and isolating the lines where an addition is made, ignoring deletion lines. An

example of a body from a Git diff is shown in Listing 4.1. By only focussing on the added lines, we train the model only on data that is actually pushed to the repository.

Listing 4.1: Git diff body snippet from commit '8d3b424'

```
    def get_data_home(data_home=None) -> str:
         data_home = environ.get('SCIKIT_LEARN_DATA',
                                 join('~', 'scikit_learn_data'))
         data_home = expanduser(data_home)
-        if not exists(data_home):
-            makedirs(data_home)
    ...
+        makedirs(data_home, exist_ok=True)
         return data_home
```

4-2 Formatting

After pre-processing the clean data, the next step is to get the code in a format that can be interpreted by our training pipeline. To train a Named Entity Recognizer, we need both the tokens extracted from the commit and their respective tags, indicating the category of the token. In our proposed models, we use the training format called 'CoNLL-2003' [20] which has been used in previous work for several downstream natural language tasks incuding Named Entity Recognition. This format consists of two columns and has one entry per token in the data set. Using the built-in Python tokenizer to tokenize Python code, we get all the tokens in a Python snippet and use those as the first column. The second column consists of the label of the token. This can be an 'O' which indicates the absence of an entity or can be the label with a 'B-' or an 'I-' in front of the label. The B indicates that the token is the beginning of the entity and I indicates that the token is inside the entity which is always after a 'B-' label. An example of sensitive data injected into Python code is given in Listing 4.2

Listing 4.2: Sensitive data injected code

```
import numpy
John Doe
def ...
```

In this case 'John Doe' is a person and needs to be labelled in the CoNLL2003 file. While 'import numpy' would appear in the file with 'O' assigned to it:

```
'import - O'
'numpy - O'
'John B-PERS'
'Doe I-PERS'
'def - O'
...
```

Indicating that 'John Doe' is the full PERS entity and 'John' is the beginning of the PERS entity and 'Doe' is inside of this entity.

4-3 Injecting sensitve data

Although we want to train our model in code and not in free text, the context does not generally matter that much in the fine-tuning process which allowed us to inject sensitive data at random. After every token of clean data, there is a 10% chance of injecting some sensitive data. The different classes of sensitive data we want to inject have the same probability to be injected and are evenly distributed (roughly the same support for every class except for 'O'). Our model is trained with the following entities of sensitive data, which are relevant to the domain in which ING operates - i.e., fintech:

1. API and secret keys
2. Personal names
3. Organisations
4. Geolocations
5. IBANs
6. Websites
7. Emails

4-3-1 Sources for sensitive data

Synthetic data

As we can not actually use real IBANs, Emails and personal information like names and addresses to train our model, we generate synthetic data. To do this, we use 'Faker' which is a Python package that generates fake data [21]. We used Faker to generate IBANs, emails, locations, website URI's and personal names in many different languages such as Dutch and English.

Public Organizational data

The '7+ Million Company Dataset' was published by the 'People Data Labs' on Kaggle and contains more than seven million company names[1]. From this large data set we extracted the company names. When we want to inject a company name we selected a record from this data set.

Regular expressions

If you want to detect API and secret keys in code, you need a lot of examples to train our model on. As we can't get so many API keys from the providers we had to be a little bit clever about it. We took the list of regular expressions as mentioned in section 4-6 and generated random string examples based on the regular expression using the 'Xeger' library available on GitHub[2].

[1] https://www.kaggle.com/datasets/peopledatalabssf/free-7-million-company-dataset
[2] https://github.com/crdoconnor/xeger

4-3-2 PII in the data set

Table 4-1 contains the entities and their occurences in the final data set:

Entity	Support
EMAIL	1382
IBAN	1383
KEY	1314
LOC	1324
NAME	1322
WEBSITE	1394
Total	8119

Table 4-1: The number of occurences per entity

4-4 Model training

State-of-the-art Transformers models such as GPT-3, BERT and RoBERTa have a lot of parameters to optimize their performances. For the experiments we use a RoBERTa-like model trained on the CodeSearchNet dataset from GitHub [22] called ‘CodeBERT’ [23]. The specifications of the training machine are in appendix A-1.

4-4-1 Ktrain

We use ‘Ktrain’, a very popular lightweight wrapper for the deep learning library TensorFlow Keras (and other libraries), used to build, train, and deploy neural networks and other machine learning models [24]. This tool allows for quick and easy training on custom data sets on any of the models available in the Huggingface library.

Contributions

Initially, Ktrain’s Named Enitity Recognition features were only available for BERT and DistilBERT versions of Transformers as the tokenization and embedding functions were specifically set up for the syntax of these models. As CodeBERT is a RoBERTa-based model, the tokenization and embedding functions had to be altered to allow for training using a RoBERTa-based model. With some helpful guidance of the owner of the repository we created a generalized solution to the functions so that all language models can be used when training a Named Entity Recognizer (the process can be seen in issue #437). Our solution is added in version 0.31.x of Ktrain through PR #441.

An important metric that we want to collect is the confidence score that a word is indeed a valid entity. However, Ktrain had mutual exclusivity regarding generating confidence scores and merging tokens to words as there was no functionality yet to calculate a confidence score for the merged tokens. Here is an example using a random generated IBAN:

word = "NL49INGA0684739664"

tokens = [(NL, 0.99), (49,0.94), (INGA, 0.96), (0684739664, 0.91)]

merged = (NL49INGA0684739664, 0.95)

We created a PR in which a function was proposed where the mean of the confidence scores of the tokens was calculated and used as the confidence score for the merged tokens. This can be seen in PR #445 which has been accepted and merged into main.

4-5 Validation

The model was validated using the following approach:

- The Scikit-learn data set was split into training and testing sets of respectively 80% and 20%. The reason we did not use Scikit-learn for validation is that we use a different Python repository, 'Flask', to validate our model on. Next to Flask, we will also use a JavaScript and C++ repository, 'React Date Picker' and 'ImGui', for validation. This way, our model is validated with no bias regarding the Scikit-learn repository. For tests including hyper parameters or between-models, we stick to the performance metrics of our test set.
- For calculating the quality of a model we create a classification report using Scikit-learn's metrics library. The score we are comparing between models is the F_1 score:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

When training multiple models of which we measure and compare the quality, we use the same training and test data files.

Next to the overall quality of the model, another very important metric for this specific use-case is the precision. The aim of our tool is to detect entities in committed code and have the developer check the commit if it is flagged by the tool because it contains sensitive data. If our model often detects a significant number of false positives, the developer would see his or hers time wasted. This would lead to the developer to decide that it is easier to just ignore the warnings and not check his committed code risking a leak of actual sensitive data. The precision is calculated as follows:

$$\text{Precision} = \frac{\#\text{ of True Positives}}{\#\text{ of True Positives} + \#\text{ of False Positives}}$$

4-6 Regex & Checksum

We also want to prevent leakage of data coming from secret keys, IBANs or Email addresses. Given that these data types usually follow a well defined pattern, we can use regular expressions. Take the Amazon Web Services (AWS) issued clientID as example; it always start with 'AKIA' followed by 16 characters being numeric or alphanumeric, which can be easily identified using regular expressions. Along with checksum validation on the matched entities this will form as extra security and reduce false positives. In table 4-2 the entities we match via regular expressions are given. For matching secrets and clientIDs from API key providers we use a list of regular expressions by the state-of-the-art tool GitLeaks [13].

4-6-1 Checksums

To validate our matched entities we have several checksums:

Entropy calculation

Shannon Entropy is a measure of the degree of randomness in a set of data and can in our cased be used to determine if recognized keys are actual keys that are randomly generated by a computer or fake keys that are put in place as an example by developers. Using the entropy calculation we can thus conclude whether a recognized key pattern is a computer generated key given by an API key provider or a fake key. The algorithm to calculate the entropy is given in Listing 4.3, the higher the entropy the likelier it is that it is a random generated key.

Listing 4.3: Probability of randomness calculation

```
# get probability of chars in string
prob = [ string.count(c) / len(string) for c in
    dict.fromkeys(list(string)) ]
# calculate the entropy
entropy = -sum([ p * math.log(p) / math.log(2.0) for p in prob ])
# Gain a probability between 0 and 1
return 1.1 - (1 / entropy)
```

BSN algorithm

The BSN or 'BurgerServiceNummer' is a citizen service number and a unique personal number allocated to everyone registered in the Personal Records Database (BRP) in The Netherlands[3].

To validate a 9-digit BSN, there exists a validation method called 'elfproef' or '11-test'. The 11-test is as follows:

Let the BSN be 'ABCDEFGHI', then

`(9A) + (8B) + (7C) + (6D) + (5E) + (4F) + (3G) + (2H) + (-1I) mod 11 == 0`

If the result is not 0, the BSN is not valid and the probability is high that it is not actually a BSN.

IBAN algorithm

An IBAN is validated by turning every character in a digit and taking the remainder after doing a modular division of this number by 97.

1. Put the first four characters at the end of the IBAN.
2. Replace every character by two digits, A = 10,...,Z = 35
3. Do a modular division of this number by 97.
4. If the remainder is 1, the IBAN is validated.

If the remainder is not 1, it is likely that this is either a placed example by a developer or something else entirely.

[3] https://www.government.nl/topics/personal-data/citizen-service-number-bsn

Suffix validation

To validate both emails and websites we take the suffix of the address and run it against the 'Public Suffix List'[4] to check whether or not it exists. If it exists, there is a high probability that the website or email is indeed valid. For Emails we also check whether it contains an '@'.

4-6-2 Entities and their checksum

The entities and their checksums are depicted in table 4-2.

Entity	Checksum
API secret keys	Entropy calculation
API access keys	Entropy calculation
BSN	Validation algorithm
Emails	Validate suffix
IBAN	Validation algorithm
Website	Validate suffix
Corporate key	None

Table 4-2: Regular expression matched entities and their respective validation method

[4] https://publicsuffix.org/

4-7 Full framework

The framework consisting of the different elements to recognize the entities in code is shown in Figure 4-1.

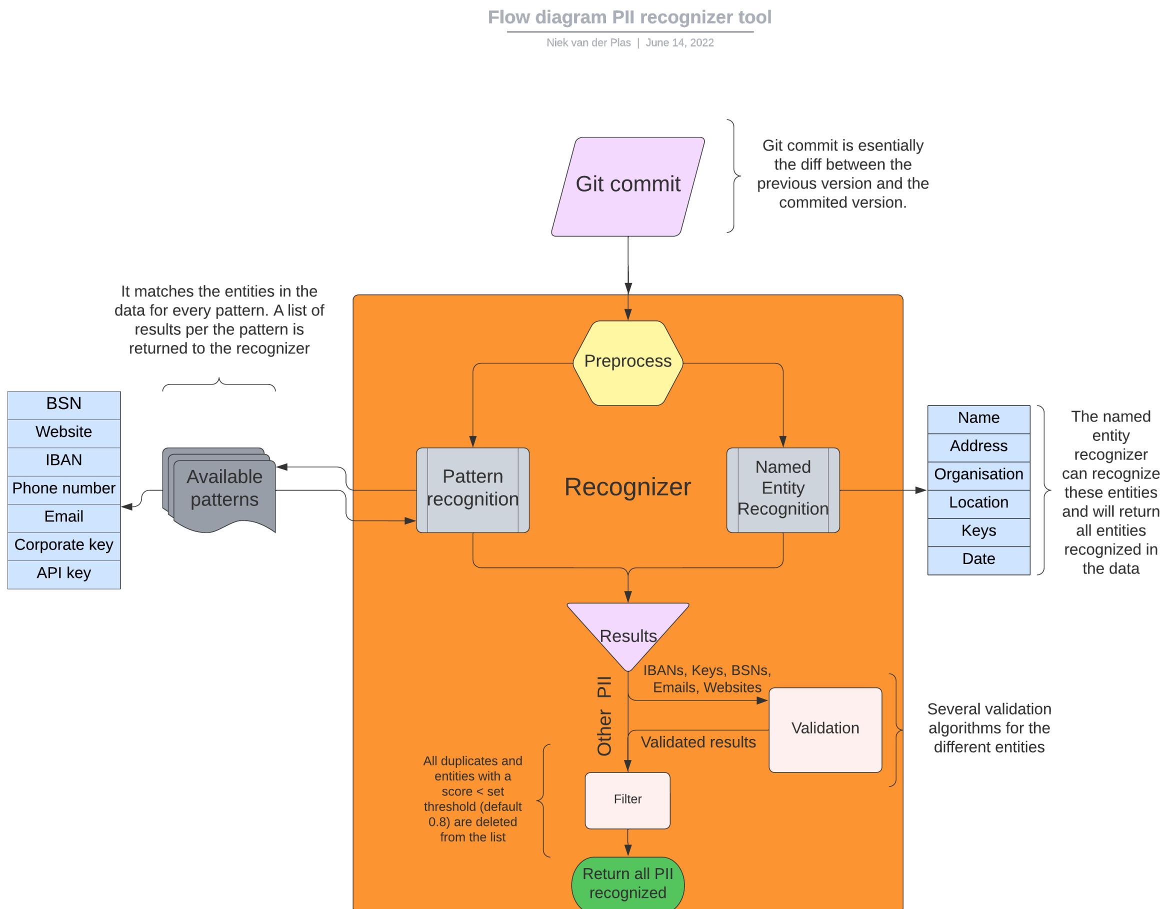

Figure 4-1: Framework of the proposed tool

The flow is as follows: First, the tool receives a Git commit which contains a diff. The Python files in this diff get pre-processed to filter out all the addition lines. These lines then get fed to both the pattern recognition and NER module where it is analyzed for entities as depicted in the framework. The results then get combined after which the entities that can be validated, are. If an entity is detected by both pattern recognition and Named Entity Recognition, either of them is discarded. Entities that are invalid, receive a confidence score of 0.1. All entities with a confidence score lower than the set threshold are then discarded. The threshold is set by default to 0.8 as most false positive results fall below that score making it a good threshold. After filtering the results, the list is returned.

Chapter 5

Results

As our goal is to create a PII detection framework containing a Transformer model which should perform accurately and have a minimum number of false positives, it is important to tune the parameters of this model to the most optimal values. To determine these optimal values several experiments were executed. We also compared RoBERTa and CodeBERT to see if pre-training on code makes a difference to the quality of the model. We continue with validating our model after which we use our complete tool to scrape a repository for entities over a period of time.

5-1 Hyper parameters

To achieve a model with a good generalization and of high quality, we focus on three important hyper parameters as initially mentioned in the original BERT paper by Devlin et al. [8]. They mention that for fine-tuning, most model hyper parameters are the same as in pre-training, with the exception of the batch size, learning rate, and number of training epochs. These experiments are done solely with the training and test data set without the use of the validation set as we just want to optimize the parameters of the model for our data.

5-1-1 Learning rate

The learning rate controls how quickly the models adapts to a problem and is used to update the weights of a model in a stochastic gradient descent algorithm. A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution and when it is too small the training can get stuck and move nowhere. Determining the optimal learning rate for the model can be challenging and should be carefully selected. Luckily, Ktrain provides a very elegant function in which we can track the loss of the model as the learning rate is increased to determine the optimal learning rate for our model, see Figure 5-1.

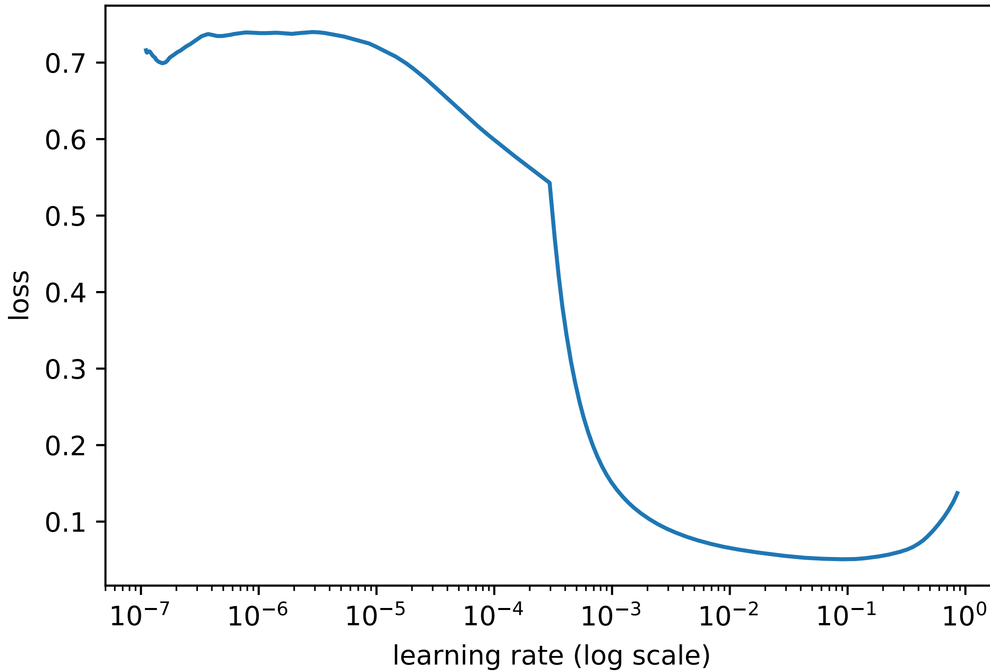

Figure 5-1: Learning rate plot

According to the documentation of the function, the maximal learning rate still associated with a still-falling loss (prior to the loss diverging) is the ideal learning rate for our model. Based on our plot, 0.005 (or 5e−03) is the optimal starting learning rate.

To verify that 5e−03 is indeed the optimal learning rate for our model, we fine-tuned the CodeBERT model using different starting learning rates to gather results on the various entities it predicts. The experiments were done with a batch size of 64 and 2 epochs. Table 5-1 shows the F_1 scores for each of the entities per tested learning rate, as well as the weighted average. It is important to note that while another learning rate might achieve a higher F_1 score, this rate could be suboptimal as it can be that the model does not generalize well beyond the training and test data and is thus of no use when introduced to a new context.

Entity	1e−1	5e−3	2e−3	5e−4	1e−4
Email	0.00	0.99	0.99	0.98	0.92
IBAN	0.00	1.00	1.00	1.00	0.93
Key	0.00	0.99	0.99	0.97	0.80
Location	0.00	0.95	0.95	0.86	0.55
Name	0.00	0.96	0.96	0.83	0.60
Organization	0.00	0.89	0.89	0.80	0.53
Website	0.00	1.00	1.00	0.99	0.98
Weighted avg	0.00	0.97	0.97	0.92	0.76

Table 5-1: F_1 scores of the entities on different learning rates

As the results in table 5-1 show, a learning rate of 5e−03 has high F_1 scores overall, and is thus chosen over 2e−03 as the ideal learning rate for our data to lead to the highest quality model without poor generalization, as the graph in Figure 5-1 already predicted. This value was selected as the default learning rate.

5-1-2 Batch size

Batch size is another very important hyper parameter to tune in modern deep learning systems. The batch size defines the number of samples that will be run through the model per iteration [25]. For example: 17000 samples with a batch size of 64 thus consists of 266 iterations of which 265 iterations have 64 samples and 1 iteration has 40 samples, these 266 iterations would be one epoch.

A high batch size is often desired by practitioners of machine learning as it allows for computational speedups, reducing the time required for a model to train. However, this speedup comes with a drawback: whenever the batch size is large enough, it can lead to poor generalization to the overall problem. For both ends of the spectrum, a larger or a smaller batch size, there are pros and cons. A larger batch size speeds up the training but can lead to a poor generalization and a smaller batch size leads to faster convergence to an overall good solution albeit significantly slower.

The choice for a batch size not only takes these choices in mind but also the required memory use, a larger batch size does require a lot more resources from the systems that train the model and the size is thus often limited to the memory available in the system.

Training the model has shown that the accuracy of the model remains stable while the time it takes to train is significantly faster when using a greater batch size.

Taking into account the above, we choose the largest possible batch size our machine is capable of processing memory-wise corresponding to 64.

5-1-3 Number of epochs

The last impactful hyper parameter is the number of epochs to train. An epoch is one complete pass through the entire training data set in which each sample has had an opportunity to update the internal model parameters [25]. As an epoch is just the number of iterations through the training set, it can take a long time to make a complete pass depending on the size of the data set. Furthermore, more epochs can result in over-fitting the model, which means that the model is trained on the data set too much that it becomes dependent on the data set and fails to fit 'new' data.

The number of epochs can be an integer value between 1 and infinity. As we want to limit the time it takes to complete, as well as maximise the performance of the model, we will try and evaluate different epochs starting from 1 up to the point where the increase in accuracy of the model is no longer worth the trade-off against the time it takes to train.

With the time to completion in mind along with the increase of accuracy the epoch brings (as shown in table 5-2), **two** is the ideal number of epochs for our training.

Epoch	1	2	3
Accuracy	0.94	0.97	0.97
Time (s)	706	1345	1967

Table 5-2: Accuracy and time to completion per epoch

5-2 RoBERTa vs. CodeBERT

While RoBERTa and CodeBERT tokenize their input the same way, their text embeddings are not the same because they are trained on different text corpora. RoBERTa [26], which is a robustly optimized BERT, was trained on a combination of the English `WIKIPEDIA` and `BookCorpus` [27], while CodeBERT [23], a multi-programming-lingual model, was pre-trained on Natural Language-Programming Language pairs in six programming languages. The tokens of which a segment of code are made up will thus be the same, while the way these are analysed will be different for either of the models. This is due to the fact that CodeBERT gains more knowledge initially from the context around a token as it is able to comprehend the context immediately as it was trained on programming languages.

It could be that CodeBERT performs better than RoBERTa when performing a downstream task like Named Entity Recognition already suggested by Karmakaker [12] however, the performance of RoBERTa also could be surprisingly good on this specific downstream task as the task is to identify entities which are text-based.

	precision	F_1
Email	1.00	1.00
IBAN	1.00	1.00
Key	0.99	0.99
Location	0.93	0.94
Name	0.95	0.96
Organization	0.86	0.87
Website	1.00	1.00
micro avg	0.96	0.96
macro avg	0.96	0.96
weighted avg	0.96	0.97

	precision	F_1
Email	0.99	0.99
IBAN	1.00	1.00
Key	0.99	0.99
Location	0.95	0.95
Name	0.95	0.96
Organization	0.88	0.89
Website	1.00	1.00
micro avg	0.97	0.97
macro avg	0.97	0.97
weighted avg	0.97	0.97

Table 5-3: RoBERTa(left) and CodeBERT(right) scores

As table 5-3 indicates, the difference between the quality of the two models is non-significant in terms of results on the test data. This means that even though the context of this task is largely written code, RoBERTa is still able to grasp the task at hand that is to recognize certain entities in code. While RoBERTa performs well, it still struggles with the context being written code instead of natural language, resulting in failing to recognize some (part of the) entities as can be seen in Listing 5.1 and having a lower confidence score for detected entities. The reason for this is that while the training is done in a CoNLL-2003 format which is word-for-word, actual prediction is done on whole chunks of code.

Listing 5.1: Code snippet in which entities are embedded

```
import re

class Patternmatcher:
    def __init__(self, regex = None, entities = ""):
        self.regex = regex
        self.entities = entities

    def match(self, str, skip_entities: list = []) -> list:
        if self.entities in skip_entities:
            return []
        scored_matches = []
        for regex in self.regex:
            matches = re.finditer(regex, str)
+            #For names like niek van der plas
+            #or for locations like john.doe@gmail.com
            for match in matches:
                indices = match.span()
                word = match.group()
                if word != '':
                    scored_matches.append({'entity_group': '{}'
                    .format(self.entities),'word': word,
                    'score': self.validate(word),
                    'start': indices[0], 'end': indices[1]})
        return scored_matches

    ------------------------------

RoBERTa entities = ('van der', 'NAME', 0.80543005)

CodeBERT entities = [('nicolaas van der plas', 'NAME', 0.9573172),
                     ('john.doe@gmail.com', 'EMAIL', 0.9976059)]
```

5-3 Validation on other data sets

We also generate other data sets using different libraries as to introduce another context to our model other than the Scikit-learn repository. As an extension to the detection of entities in Python code we are also curious to see how our model will perform on another programming language as an ideal model should achieve accurate results irrespective of the programming language. We choose a JavaScript and C++ repository as CodeBERT is also pre-trained on JavaScript next to Python, while C++ is not supported. This will give us another insight whether or not pre-training on multiple languages makes a difference when a model is fine-tuned on solely Python code.

We injected sensitive data into the commits at a random position in the code without breaking existing tokens. Every commit had an average of 90 tokens in it and, with 2.000 commits taken from each library, this adds up to the model classifying roughly 180.000 tokens per data set.

For these experiment we assume that the repositories do not contain any PII in its commits

and for both we choose the last 2.000 commits containing edits made in relevant files, '.py' files for Python, '.js, .jsx' files for JavaScript and '.cpp' for C++.

With our model's prediction we calculated the true and false positives as well as the false negatives as shown in Table 5-4, 5-5 and 5-6. The true negatives could not be calculated per class as we are dealing with a multi-class problem on the same data, it can be stated that a significant number of tokens were correctly filtered out as true negatives.

To calculate these metrics we keep it relatively simple:

1. Item in predicted entities but not in injected entities list → False positive
2. Item in injected entities but not in predicted entities list → False negative
3. Item in both predicted and injected entities list → True positive

It is important to keep in mind that the fine-tuned model can do one of three things. Either it can detect the whole entity, it can detect a substring of the entity or it can detect a little bit more than the entity by also including the subsequent word. The false positive cases contain thus a lot of actual injected entities of which a part is missing or have some additional data appended to it.

5-3-1 Python language

We choose the Flask repository[1] as a Python library to test our model on. As Table 5-4 shows, the model has trouble with detecting organizations and locations as can be concluded by the large number of false positive cases. While there are a number of these false positive that are actually a substring of the whole entity, a large number is not, which can be due to an organization name being relatively ambiguous resulting in import libraries sometimes being classified as sensitive data. Entities like 'Werkzeug' and 'Pallets' (which is the owner of the Flask repository) frequently occur in the list of false positives.

	True positives	**False positives**	**False negatives**
Organization	99	113	10
Name	84	6	17
Website	79	19	2
Location	92	31	16
IBAN	78	5	3
Email	79	2	3
Key	83	8	14

Table 5-4: Results of prediction on the Flask repository with injected entities

[1] https://github.com/pallets/flask

5-3-2 JavaScript language

We choose the JavaScript repository[2] as a JavaScript library to test our model on. JavaScript would be familiar to CodeBERT, as it was pre-trained on a variety of programming languages including JavaScript. As indicated by Table 5-5, the model has the same issues as with the Python language with a significant number of false positives in the organization class. While the model was fine-tuned on Python data, it benefits from the fact that it was also pre-trained on JavaScript as the results compared to Table 5-4 are similar to the results presented in Table 5-5.

	True positives	False positives	False negatives
Organization	85	112	23
Name	55	5	18
Website	64	0	14
Location	57	47	19
IBAN	64	1	11
Email	79	7	3
Key	53	7	17

Table 5-5: Results of prediction on the React Date Picker repository with injected entities

5-3-3 C++ language

We choose the ImGui repository[3] as a C++ library to test our model on. This programming language would not be familiar to CodeBERT, as it was not pre-trained on C++. Additionally, this language was not used for fine-tuning our Named Entity Recognizer. It is not surprising that the results in Table 5-6 significantly differ from the results in Table 5-4 or 5-5. While roughly 200 false positive detected entities are something along the lines of 'ImGui', 'Im::Gui', the model didn't perform as well in the C++ language. This is largely due to the fact that fine-tuning on Python language is not the same as on C++ language as the syntax in C++ is completely different than Python's with the model additionally not being pre-trained on C++. This difference most likely explains the variation in results. It is worth noticing however, that in both cases the model has a large number of false positives in either organizations or geolocations.

To actually have a model that performs well in C++, the model should be fine-tuned on git commits that contain C++ snippets of code with injected sensitive data or a corpus of C++ code should also be used for the pre-training of RoBERTa. If someone wants a model that accurately detects entities in multiple languages, it should be fine-tuned on a combination of git commits containing sensitive data in a variety of programming languages which are also supported in the pre-training of the Transformer model.

[2] https://github.com/Hacker0x01/react-datepicker
[3] https://github.com/ocornut/imgui

	True positives	False positives	False negatives
Organization	182	350	23
Name	69	22	25
Website	104	3	7
Location	93	524	26
IBAN	85	24	11
Email	91	0	13
Key	78	3	44

Table 5-6: Results of prediction on the ImGui repository with injected entities

5-4 Scraping git commits to find PII

Using our complete tool, consisting of both the NER model and pattern recognition methods, we scraped a repository's commit history with Pydriller [19] to find PII hidden in the commit history. We chose 'psf/Requests'[4] as the repository to scrape since it consists of mainly Python scripts. It is important to know that we can not measure the precision or recall of this experiment as we do not know the actual numbers of PII that are present in the commit history. Table 5-7 shows the number of PII found per category over the span of commit 'b8ba5c2' (16-04-2014) to commit 'da9996f' (09-06-2022) with some of the interesting entities with it.

Entity	Found	Example
Keys	24	key = 'some_cookie'; nonce="6bf5d6e4da1ce66918800195d6b9130d'
BSN	0	-
IBAN	0	-
Emails	49	dan.****@gmail.com (omitted for privacy); password@host.com
Location	68	Morsel; Queue; Requests; Sauce; Japan; Taiwan;
Website	301	http://google.com/mail; http://bugs.python.org/issue10272
Corporate key	0	-
Organizations	296	copyright; proxy; cookie jar; httplib

Table 5-7: Entities found in psf/requests

As can be noticed, and was already clear from subsection 5-3, there are a lot of predicted organization and location entities. Naturally, many of those aren't actual entities and are wrong which can be explained due to the ambiguity of organizational or geolocational data. However, when keeping in mind that we are analyzing roughly 4.000 commits, these detections do not occur frequently. The other entities are valid entities with a spike in the number of detected websites that can be explained when taking into account the application of this library, HTTP requests.

[4] https://github.com/psf/requests

Chapter 6

Discussion

While the results are promising, there are still many things that can be done to improve this research and proposed tool.

Using the threshold some false positives get through as the probability is not a very reliant metric currently. The number of false positives needs to be brought down to an absolute minimum as to avoid disturbing a developer or anyone who uses the tool with notifications of detected sensitive data. The tool can be further improved upon by introducing a whitelist and a voting classifier. A user could fill the whitelist with words and when the tool detects these words, it discards them from the results. The voting classifier can be used on all found entities instead of using a certain probability threshold to determine whether or not an entity should be discarded or not. The voting classifier in combination with a whitelist will reduce the number of false positives significantly resulting in a higher overall accuracy.

Right now, the fine-tuned Transformer model depends on the presence of case letters for example when predicting personal names or organizations. When taking the example of Figure 5.1 and making the first 'n' and 'p' in 'nicolaas van der plas' uppercase, suddenly the results for both models change:

Listing 6.1: Entities recognized with cased examples

```
RoBERTa entities = ('Nicolaas van der Plas', 'NAME', 0.96133)

CodeBERT entities = [('Nicolaas van der Plas', 'NAME', 0.99770653),
                     ('john.doe@gmail.com', 'EMAIL', 0.9976059)]
```

While case information can be very important in NER applications, it can be assumed that in our application the case information can be vague. In code context, the difference a cased letter makes can be anything from a function declaration to a variable name and the presence of PII in code can often also be uncased as it is not in the context of actual text making it less important for the writer to case entities like names and locations.

To make sure our model does not depend on cased input our Named Entity Recognizer could be trained using uncased input. More research needs to be done to conclude whether or not it would improve the models quality of prediction.

The way we generated API keys and secrets is also something that could be improved. Currently, we reverse regular expressions using the 'Xeger' library and use the keys generated using this method in our training. The problem is that those keys are often not valid examples of keys that would otherwise be generated by the providers. A better way would be to use actual (expired) keys for our training. Using keys generated by the providers, it is likely that the quality of the model in respect to the detection of keys and secrets would be better.

We also calculate the metrics using the test set for our hyper parameter experiments which is the existing built-in validation tool of Ktrain. While perhaps we should use the Flask validation set to calculate the metrics for these experiments it should not really matter as we just want to optimize the parameters for the model before doing more in-depth experiments where testing the model on the validation set is desired.

Calculating the number of true and false positives as well as the false negatives could have been executed more thoroughly. Currently, the metrics presented are not totally correct as some have been wrongfully classified as a false positive or false negative. We could have created a function which performed better in determining the actual true and false positives by accepting detected entities that are slightly malformed as true positives as they still are found. We could also have done a manual check over all entities detected against the list of injected entities which would have taken a while but would have given the most accurate results.

The model has some issues when predicting certain entities. Occasionally, it will detect irrelevant tokens as an organization or a location as these are quite ambiguous. These tokens cost the user time by having to check the commit and flag it as false positive eventually becoming reluctant to even check the warning if it happens more often. To counter this, we believe that with extra training on multiple libraries and some extra attention to identifying positions to inject data, we could largely diminish this issue. The improvement of accuracy regarding these two classes requires further investigation in creating the data set as well as fine-tuning the model. More research needs to be done to determine whether an approach can be designed to resolve this issue.

Chapter 7

Conclusion

This research aimed to propose a tool which could serve as a detection model for PII in (Python) code in which, for non-regular pattern entities, a Named Entity Recognition model was trained using the pre-trained RoBERTa-like Transformer model CodeBERT. This tool can be used as a replacement for the existing entropy-based model that is currently in place at ING. Based on the experiments with the Transformer model, we have achieved a well-performing model that can recognize a variety of entities in code quite accurately. In combination with the detection module using regular expressions and checksum validation, this framework tool could be of great value for the teams using DAP as a platform while pushing code to Azure DevOps within ING.

While this framework specifically targets the Dutch financial sector, more specifically the data engineering sector within ING who use Python, one could modify the framework to be an all-round well-performing solution for a variety of businesses in a variety of countries. While existing solutions are heavily reliant on regular expressions to detect entities, our solution is not and can detect entities that do not follow a certain pattern, like names and organisations, and can readily be trained to detect more entities in different programming languages if fine-tuned on these languages. A model fine-tuned on Python code can detect PII in different programming languages if and only if this language is used when pre-training the model (such as Go and JavaScript).

Our model performs well on existing open source Python projects with a lot of actual results answering the question that we can use synthetic data to train models that identify PII in code commits. However, as the model currently predicts irrelevant tokens as entities in the organization and location class occasionally, further investigation on fine-tuning codeBERT is needed.

Appendix A

Appendices

A-1 Specifications

- Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz
- NVIDIA GeForce GTX 1070 Ti
- 16GB DDR4 RAM @ 2133MHz in 2 slots
- Windows 11 Pro x64

Bibliography

[1] *Jumpcloud api key leaked via open github repository*, https://hackerone.com/reports/716292, Accessed: 22-01-2022.

[2] *Reviewing the 2021 united nations data breach*, https://blog.gitguardian.com/united-nations-databreach-jan/, Accessed: 22-01-2022.

[3] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[4] S. Garfinkel *et al.*, *De-identification of Personal Information:*. US Department of Commerce, National Institute of Standards and Technology, 2015.

[5] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? characterizing secret leakage in public github repositories.," in *NDSS*, 2019.

[6] *Removing sensitive data from a repository*, https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/removing-sensitive-data-from-a-repository.

[7] *About secret scanning*, https://docs.github.com/en/code-security/secret-scanning/about-secret-scanning.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[9] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[10] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[11] J. Li, A. Sun, J. Han, and C. Li, "A survey on deep learning for named entity recognition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 50–70, 2022. DOI: 10.1109/TKDE.2020.2981314.

[12] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1332–1336. DOI: 10.1109/ASE51524.2021.9678927.

[13] Z. Rice, *Gitleaks*, https://github.com/zricethezav/gitleaks, 2018.

[14] L. Geng, L. Korba, X. Wang, Y. Wang, H. Liu, and Y. You, "Using data mining methods to predict personally identifiable information in emails," in *International Conference on Advanced Data Mining and Applications*, Springer, 2008, pp. 272–281.

[15] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, "Detecting and mitigating secret-key leaks in source code repositories," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 396–400. DOI: 10.1109/MSR.2015.48.

[16] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984. DOI: 10.1109/TSE.1984.5010248.

[17] *Ing at a glance*, https://www.ing.com/About-us/Profile/ING-at-a-glance.htm, Accessed: 01-03-2022.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2018. DOI: 10.1145/3236024.3264598.

[20] E. F. Sang and F. De Meulder, "Introduction to the conll-2003 shared task: Language-independent named entity recognition," *arXiv preprint cs/0306050*, 2003.

[21] D. Faraglia and Other Contributors, *Faker*. [Online]. Available: https://github.com/joke2k/faker.

[22] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search," *arXiv:1909.09436 [cs, stat]*, Sep. 2019, arXiv: 1909.09436. [Online]. Available: http://arxiv.org/abs/1909.09436 (visited on 03/12/2020).

[23] Z. Feng, D. Guo, D. Tang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[24] A. S. Maiya, "Ktrain: A low-code library for augmented machine learning," *arXiv preprint arXiv:2004.10703*, 2020. arXiv: 2004.10703 [cs.LG].

[25] J. Brownlee, "What is the difference between a batch and an epoch in a neural network," *Machine Learning Mastery*, vol. 20, 2018.

[26] Y. Liu, M. Ott, N. Goyal, *et al.*, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[27] Y. Zhu, R. Kiros, R. Zemel, *et al.*, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 19–27.