Università
della
Svizzera
italiana

Bachelor Thesis

Experimenting with Curriculum Learning in Software Engineering on code-related tasks with codeBERT

Faculty of Informatics
June 23, 2023

Abstract

Deep neural networks, including Transformers, can be trained to learn almost any task by looking at a large set of supervised instances. When considering software engineering, there are many tasks which can be automated, fully or partially, by employing deep learning. For instance, mundane tasks such as bug-fixing or writing documentation are considered low-creativity and time-consuming. The result of these tasks is derived from the underlying, pre-written, code. This code can be fed into a deep learning system as a supervised instance. Several deep learning models exist that specialize in mentioned code-based tasks, however the performance is still limited. To this aim, one concept that has been successfully applied in other domains (other than software engineering) is that of curriculum learning. The methodology is similar to the way humans and animals learn by tackling simpler problems before graduating to more complex. By employing curriculum learning, we leave the majority of the system as it is, and we alter the way in which data is fed to the system. The input stream to the deep learning system is sorted based on a predefined complexity/simplicity metric and separated in N batches. The model is then fine-tuned on each of the N batches consecutively. This should help the model generalize better as it learns gradually, rather then all-at-once. This methodology proved partially successful. When we tested the curriculum learned model compared to the benchmark codeBERT model, we achieved both positive and negative results depending on the tackled task.

Author: Fazlic Mak
Advisor: Prof. Dr. Bavota Gabriele
Co-advisor: MSc. Mastropaolo Antonio

Contents

1 Introduction

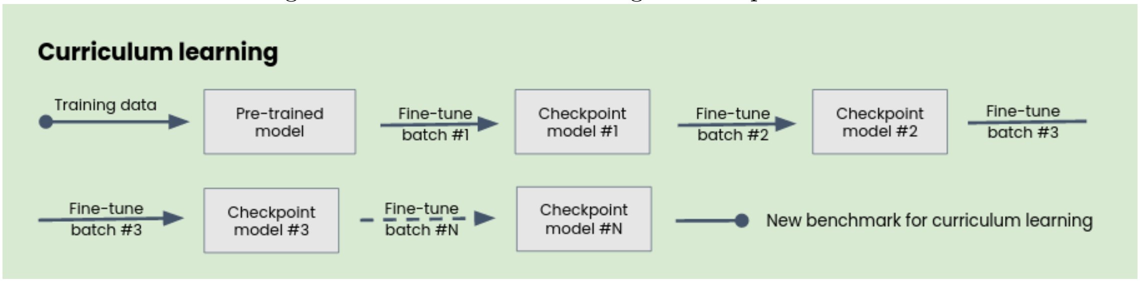
When it comes to software engineering, it is crucial to keep readability, maintainability and performance of the code high. There are many techniques, best-practices and tools within the industry to make sure the software engineering process performs. One of the domains that has been increasingly employed to provide support to software engineers is machine learning. In recent years this was expanded to deep learning. Deep learning is useful as a lot of the code that is being written every day can be stored and used as supervised instances in training a deep learning model. Therefore, post-training we have a model that has generalized the context, extracted features from the code and could be used for the generation of similar code or other code related tasks.

Transformer models can be used to train a highly accurate machine learning system based on the input tokens. These tokens can represent a wide variety of data types and structures, therefore such models are effective in all kinds of applications. They were first introduced in 2017 in a paper "Attention is all you need" [1] and quickly after they replaced RNNs [2] and LSTMs [3] in Natural Language Processing tasks and similar text-dependent tasks. In the context of this project we use a transformer model which was previously pre-trained on source code, and we further add more data to improve its performance in fine-tuning. That is to say we take a pre-trained transformer model and adapt it to our specific task, by using the data we wish the model to be specialized in recognizing and managing.

The pre-trained model we are fine-tuning is nicknamed "codeBERT" [4]. It is a deep learning model developed in 2020 by Microsoft that was trained on code from GitHub repositories in 6 different programming languages (Python, Java, JavaScript, PHP, Ruby, Go). The training was conducted similarly to that of a multilingual BERT (Bidirectional Encoder Representations from Transformers) [5] that was pre-trained by Google on an unlabeled, text data from 104 different languages. Similar approach was taken by Microsoft for "CodeBERT", but with the usage of code. The model is open source and available in the GitHub repository to be used or further fine-tuned.

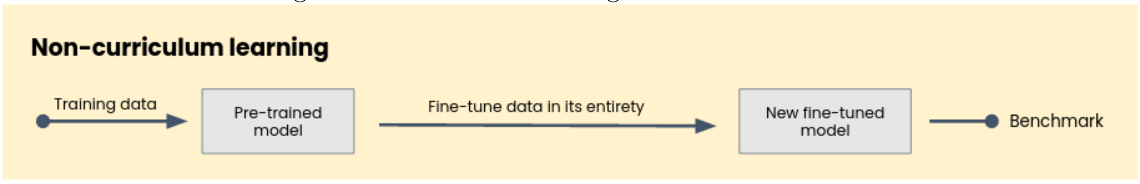
The focus of the research is to push the performance of fine-tuning codeBERT for code related tasks by not changing the dataset, but by employing an alternative learning strategy called "curriculum learning". We investigate the performance of curriculum learning against the benchmark that was provided by Microsoft to ultimately conclude on the effectiveness of the strategy. Curriculum learning in deep learning is based on passing the data to the system sorted based on a predefined complexity/simplicity metric and separated in N batches. The model is then fine-tuned on each of the N batches consecutively as illustrated in figure 1.

Figure 1: Curriculum fine-tuning on multiple batches



This should help the model generalize better as it learns concepts gradually from simple ones to complex ones, rather then all-at-once as illustrated in figure 2.

Figure 2: Classical fine-tuning on the whole dataset



Furthermore, it is simple to evaluate as the strategy is based on prepossessing the data, rather then changing the underlying deep learning architecture.

The research task consists of two sub-tasks, automatic bug-fixing and automatic documentation generation. For both tasks we have codeBERT benchmarks and we will use codeBERT as base to fine-tune on code blocks which represent methods. The programming language we want to fine-tune on is Java. The instances of Java code were mined from GitHub commits performed by software engineers, and when it comes to the dataset for bug-fixing specifically, the buggy methods committed are paired with the fixes in the commits which followed. For bug-fixing we have a total of 123,804 supervised instances split in train, test and validation sets of both small and medium size, while for automatic documentation generation or code summarization we have 164,923 supervised instances split into train, test and validation sets.

After conducting the pre-processing and employing curriculum learning, the results were partially positive. Although there was an improvement on the benchmark, this improvement was limited. This was expected considering we are not changing the underlying architecture of the deep learning system. However, we also evaluated the trained models on several other metrics and found that the curriculum learned model produces predictions which for one task perform better than benchmark when evaluated against the test set.

1.1 Report Structure

The following report is structured as follows:

- Introduction
- Background and State-of-the-art
- Approach
- Study Design
- Results Discussion
- Conclusions and Future Work

2 Background and State-of-the-art

In light of our investigation's focus, we initially present background information about curriculum learning. Following that, we delve into a discussion about the related literature of code-realted tasks mostly relevant to the two tasks we're targeting for our experiment, namely automatic bug-fixing and code summarization.

2.1 Curriculum Learning

Curriculum learning is an innovative methodology that can be applied to machine learning and other domains. This approach draws its inspiration from educational methodologies and curriculum design [6]. The concept revolves around introducing training data to a deep learning algorithm in a progressively increasing complexity. This involves arranging the training set instances based on their difficulty or complexity. After establishing the level of complexity, considering the nature of the task at hand, the learning algorithm can progressively receive data. This involves systematically escalating the complexity of the samples provided to the deep learning model during the learning procedure.

By leveraging such a methodology during the learning procedure our objective is to improve the performance of the deep learning system by enhancing its ability to generalize effectively for a given task. Due to the computational complexity and variance in the output of in our difficulty/complexity function, we can deploy curriculum learning by simply classifying data, rather then fully sorting it. First example of this is a paper published in 2009 called "Curriculum Learning" [7] where the authors applied gradual increase in complexity of shapes and figures they were trying to get recognized by deep deterministic and stochastic neural networks. The research yielded a successful outcome, leading to a subsequent surge of research that expanded upon the concept of curriculum learning in various domains.

With the recent advancements achieved by incorporating curriculum learning (CL) into the training of deep neural architectures, researchers have begun exploring whether this training approach could be beneficial for code-related tasks. For instance, in "Recent Advances in Neural Program Synthesis" [8], Kant et al. investigate the application of CL in the context of program synthesis where a user would specify a task to be completed in a given programming language and the model trained with curriculum learning would complete the code. Following a similar approach to the aforementioned authors, our investigation aims to explore the potential utility and scope of this straightforward yet effective methodology in the realm of deep learning solutions tailored for software engineering related activities. Specifically, we focus on two code-related tasks: **(i.)** automatic bug-fixing and **(ii.)** code summarization.

2.2 Automating Code-related Tasks Using Deep Learning

Prior to the widespread adoption of neural model architectures, the automation of code-related tasks predominantly relied on relatively simplistic approaches that required expert knowledge or familiarity with the programming language or style. These approaches included meta-heuristic algorithms such as a Genetic [9] algorithm that heavily relied on randomness to generate and evaluate new instances. Additionally, traditional machine learning algorithms such as SVMs [10] and K-means [11] were also utilized. While these techniques demonstrated good performance for specific tasks, they suffered from two major limitations: **(i.)** decreased performance when applied to large datasets, and **(ii.)** reliance on static code analysis, which is limited by the expertise and underlying algorithm used.

On the other hand, the advent of neural architectures has revolutionized the automation of code-related tasks like code completion, bug-fixing, and code summarization, among others. This advancement has taken these activities to unprecedented levels of automation. Notably, a groundbreaking work in automatic code generation is the DeepCoder [12] project, which aims to generate code automatically through deep learning techniques. White et al. introduces a deep, compositional, learning-based detection approach that can identify code clones by inducing representations at various levels of granularity. A common thread among many related works, is the utilization of a specific deep learning architecture known as the Long-Short Term Memory (LSTM) [3] model.

LSTMs, are a specific type of Recurrent Neural Networks (RNN) [2] that work by maintaining an internal memory to process sequential information, making them suitable for code-related tasks. However, the recent introduction of Transformer models has overshadowed the usage of RNNs/LSTMs. Transformer models have emerged as the new state-of-the-art for various code-related activities, including automatic documentation of code components, code generation, and code translation. This is possible thanks to the attention mechanism model that would allow to capture long-range term dependencies in the input by focusing on different parts of the input, capturing meaningful relationships and dependencies, thus making it highly effective when dealing with code-related tasks.

2.3 Automated Bug-fixing

Automated bug-fixing is defined as an unsupervised change made to a given code block for a purpose of increasing correctness. A common technique is called generate-and-validate where the bug-fixing solution generates several blocks that could potentially solve the issue at hand. Afterwards, these blocks get validated against the test cases specified by engineers. The best performing blocks are candidates for the replacement of the initial buggy code. To generate the blocks for validation we can employ several techniques such as code-mutations, predefined templates, program synthesis, etc... An application of such techniques can be found in “CodeHint: dynamic and interactive synthesis of code snippets“ [13] which focused on generating code snippets of Java code to boost programming efficiency.

Approaches to automatically produce fixes for buggy code components have evolved along with the usage of learning-based solutions such as DL-models. Such models can learn from human written patches trying to replicate the needed code changes to fix the bug, when provided as with buggy components never seen before. An example of such a technique deployed successfully was “Getafix: learning to fix bugs automatically“ [14] published and used by Facebook since 2019. Another work that coincides with our research is the “Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation“ [15] by Tufano et al, where deep learning approach was applied in the context of bug-fixing. With the widespread use of transformer models in other domains, researchers have, thereafter, started to investigate the usage of transformers in bug-fixing as well, see “Studying the usage of text-to-text transfer transformer to support code-related tasks“ [16] by Mastropaolo et al.

2.4 Code Summarization

Code summarization, or in other words creating textual summaries of the code for documentation purposes, is a problem in natural language processing where we aim to generate human readable text. It overlaps with software engineering as the input is a function or any other block of code. Since writing documentation is an integral part of the software engineering process, automation of such a task is time-consuming and can be automated as proven by Huang et al. in “Learning Human-Written Commit Messages to Document Code Changes“ [17].

During the infancy of code summarization as a field, researchers relied on heuristics and rule-based approaches to generate documentation automatically for code components. For example, Moreno et al. in the paper “JSummarizer: An automatic generator of natural language summaries for Java classes“ [18] utilized a predefined set of templates to generate natural language summaries for Java classes. Often in code summarization the NLP solution relies heavily on ASTs (Abstract Syntax Trees) which represent the underlying code structure which the algorithm should describe. In recent times, the adoption of RNN/LSTM models initially and later transformer models has resulted in significant advancements in code component summarization. These models have demonstrated an improved performance and effectiveness in summarizing code. Example of LSTMs employed to generate documentation is “Deep Code Comment Generation“ [19] in 2018, while on the other hand we have “A neural attention model for abstractive sentence summarization“

[20] which uses RNNs to accomplish the same goal, but for any sequence of characters. Example of deep learning on transformers applied for code summarizaion would be a paper "An Empirical Study on Code Comment Completion" [21] by Mastropaolo et al. where Text-To-Text Transfer Transformer (T5) [22] architecture is used in auto-completing a code comment the developer is typing. CodeBERT [4] is an example of code summarization applied successfully and was used as a benchmark and a starting point for this study.

3 Approach

This section presents the necessary information for reproducing the experiments. In this context, we begin by discussing the dataset used for the experimentation and the methodology employed to evaluate the curriculum learning approach on the two suggested code-related tasks. The objective is to enhance the efficacy of DL-based solutions in automatically producing fixes for buggy code code and generating summaries of code components.

3.1 Defining CL for Bug-fixing

In the context of bug-fixing, we used a pre-trained model as a base and applied curriculum learning while fine-tuning for Java mathods. The integral part of curriculum learning, as a pre-processing technique, is the way we set up the data classification process. We have to decide on two things. **(i.)** The complexity function that as input takes a supervised instance and provides us with the output of a complexity metric. **(ii.)** Manner in which the classification is conducted based on the complexity metric.

(i.) When it comes to the complexity function, for this task, we decided to use a carefully modified Levenshtein [23] distance. In order to measure how different, or in other words buggy, the code we are presented with actually is, we need a numerical value that represents the difference between the buggy and the fixed Java method. We decided to count the number of insertions, deletions and modifications needed to convert the buggy instance to the fixed one. This was done on the level of tokens, which we defined as any combination of characters deliminated by spaces or tabs. Once we defined our complexity function we ran it over all supervised instances in our dataset. Once the function reads both methods as strings (buggy and fixed), it converts them into tokens, counts the number of token changes needed to convert the buggy string into the fixed one, and returns the integer representation of this distance.

(ii.) Now that we have a direct mapping from the code instance pair (buggy and fixed) to integer representing the defined complexity, we can group those instances into complexity-alike classes. These classes would be later used in increasing order of complexity as input to the fine-tuning algorithm as defined by curriculum learning. We decided to go with four classes, being “low complexity“, “low-medium complexity“, “medium-high complexity“ and “high complexity“.

In order to classify an instance, we had to define the cut-off ranges for each class. For this we used statistical analysis of all Levenshtein distances produced and determined that the medium sized instances should be contained in the IQR, with the median splitting the medium-low and medium-high classes. For the low complexity instances we used instances with Levenshtein distance below the first quartile, while for high complexity we used insances with Levenshtein distance above the third quartile. It is important to note that once we had the classification completed and the instances separated into four batches, we decided that a given batch should include the instances from the batch with lower Levenshtein distance. That is to say, for N batches, if we consider a given batch n_i, it includes all instances from the batch n_{i-1}, while naturally including its own instances. Such a configuration results in the four batches of increased complexity having a cardinality of 25%, 50%, 75%, 100% of the total dataset respectively.

3.2 Fine-tuning for Bug-fixing

When it comes to bug-fixing, we used the “code-refinement“ section of the CodeXGLUE project supported by the “Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation“ [15] study.

3.2.1 Dataset

The data and the pre-trained model are already provided in the mentioned repository. The data is composed of *small* and *medium* Java methods, classified by their length. These Java methods were collected by accessing “every public GitHub event between March 2011 and October 2017“ and by collecting “commits having a message containing the patterns: (“fix” or “solve”) and (“bug” or “issue” or “problem” or “error”)“, as outlined in the mentioned paper. The total number of instances we have at our disposal depending on the size of the function, being medium or large, is presented in the figure 1. It is important to note that we also introduced a third dataset which we called “merged“ or “small+medium“. This is a simple combination of the small and the medium dataset.

Block	**Small**	**Medium**	**Small+Medium**
Train	46,680	52,364	99,044
Valid	5,835	6,545	12,380
Test	5,835	6,545	12,380

Table 1: Dataset size for bug-fixing

3.2.2 Metrics

Now that we have the data, we can apply the fine-tuning process and evaluate it, so that the curriculum learned model had a benchmark to compare against. We proceeded with the inference to generate instances from which we will compute the desired metrics. The inference was naturally done with the same hyperparameters as the fine-tuning. Metrics we decided to use for this task are BLEU [24] and Accuracy [25]. BLEU-4 is a widely used metric in deep learning NLP-based systems that measures the similarity between a predicted and the true label based on n-gram precision (in this case 4-gram). Accuracy is simply the ratio between correct predictions and all the considered samples.

The **BLEU** score is defined as follows:

$$\text{BLEU-4} = \text{BP} \times \exp\left(\frac{1}{4}\sum_{n=1}^{4}\log\left(\text{precision}_n\right)\right)$$

Where BP is:

$$BP = \min\left(1, \frac{\text{output length}}{\text{reference length}}\right)$$

And precision_n is defined as:

$$\text{precision}_n = \frac{\text{Count of n-gram matches in candidate and reference}}{\text{Count of n-grams in candidate}}$$

The **Accuracy** has the following definition:

$$\text{Accuracy} = \frac{\text{Number of correctly predicted samples}}{\text{Total number of samples}}$$

We will discuss the values obtained by these metrics in the *results* section 5 of the report.

3.2.3 Hyper-parameter Tuning and Model Validation

In order to control our fine-truning procedure, we have to specify a set of hyperparameters for the training process. For this task the list of the hyperparamaters that produced the best performing model can be seen in the figure 2.

Hyperparameter	**Value**
model_type	roberta
model_name_or_path	microsoft/codebert-base
config_name	roberta-base
tokenizer_name	roberta-base
max_source_length	256
max_target_length	256
beam_size	1
train_batch_size	4
eval_batch_size	4
learning_rate	5e-5
train_steps	100000
eval_steps	5000

Table 2: Fine-tuning hyperparameters bug-fixing

We also employed the early stopping technique to avoid overfitting. This will be further discussed in the following section 3.2.4. After running the fine-tuning with the specified hyperparameters on local GPUs, we saved the fine-tuned model to be used for inference.

3.2.4 Early Stopping

Another technique we applied in the fine-tuning process is early stopping. This was done to avoid overfitting and reduce train time. It is a way to tell the deep learning algorithm to stop based on a given condition. We evaluate if the early stopping condition is met after each predefined time interval. In our case the condition we used relied on the relative change, reflected in the performance metrics, of each time we considered to cease the run-time of the algorithm. Specifically, we evaluate the model every N training steps, and check if there was an improvement in the last k evaluations. For this tasks we used $k = 5$, such that we run the fine-tuning on the entirety of the dataset or a given batch, while keeping track of the best evaluation cycle (every N steps) and save the model associated with this checkpoint. Then, if there were no improvements after 5 evaluation cycles, we consider the checkpoint model for inference or as a starting point to fine-tune the consecutive batch. For bug-fixing, we implemented the evaluation of the best potential checkpoint after every 5000 training steps.

3.3 Defining CL for Code Summarization

In the context of code summarization, we used a different pre-trained model as a base and applied curriculum learning while fine-tuning for summariazion of Java methods in natural language. Once again we had to define two things in order to set up curriculum learning for this task. **(i.)** The complexity function that as input takes a supervised instance and provides us with the output of a complexity metric.

(ii.) Manner in which the classification is conducted based on the complexity metric.

(i.) As a complexity function for this task, we used a simple length method. For this task the data was already tokenized and we deiced to use the token definition used in the application of CodeSearchNet [26] instances in CodeXGLUE. Considering the methods were presented in the tokensized form, we were able to count the number of tokens for each method and use that as our complexity metric. The intuition behind this decision was the possible correlation between the length of the Java method and it's respective documentation. In other words, the length of summary is relative to the length of the method, so if the Java method is short, the summarized documentation of that function should follow to be short, and vice-versa. This has shown to be a simple and effective way to implement the complexity function for this task.

(ii.) Once the complexity function was defined, we had to classify the instances by defining the cut-of ranges in the same way as we did for bug-fixing. In fact, the procedure for this task was the same as bug-fixing with the four classes being "low complexity", "low-medium complexity", "medium-high complexity" and "high complexity", the medium complexity instances represented by the IQR, seperated into low-medium and medium-low by the median. The low clomplexity class is below the first quartile and the high complexity is above the third quartile. We also decided to keep the contents of each batch in the following batch with increased complexity, as we did for bug-fixing.

3.4 Fine-tuning for Code Summarization

For code summarization, we used the "code-to-text" section of the same "CodeXGLUE" project. Once again the fine-tuning process was straightforward, due to the most of the implementation being contained and documented within the project itself. The data and the pre-trained model are already provided in the mentioned repository.

3.4.1 Dataset

Fine-tuning was conducted on the provided data which comes from CodeSearchNet [26] and was filtered to not include code with the following criteria: cannot be parsed into an abstract syntax tree, number of tokens of documents are less than 3 or more than 256, contain special tokens, documents are not English. Considering the scope of this research we used only Java code from the provided dataset. The number of Java method instances can be seen in the following figure 3 bellow.

Block	**Instances**
Train	164,923
Valid	5,183
Test	10,955

Table 3: Dataset size for code summarization

3.4.2 Metrics

Initially for code summarization we used the same metrics as for bug-fixing, being BLEU [24] and Accuracy [25] which we defined in *bug-fixing metrics* section 3.2.2. However, we got curious, regarding the possible additional insights that other metrics might bring for this specific task. That said, we decieded to use two additional metrics for this task, namely ROUGE-L [27] and METEOR [28]. METEOR is a metric that uses precision, recall and weight components to compute a score between 0 and 1. This metric is also based on the synonym matching to better understand the context of the NLP task. ROUGE-L is a way to measure the longest common sub-sequence and compare the two over the predicted and the true labels of the data. Here are the definitions of the two new metrics introduced.

The **ROUGE-L** is defined as follows:

$$\text{ROUGE-L} = \frac{\text{Longest Common Subsequence (LCS)}(\text{Reference}, \text{Candidate})}{\text{Reference Length}}$$

The **METEOR** score is defined as follows:

$$\text{METEOR} = (1 - \text{Penalty}) \times \left(\frac{\text{Precision} \times \text{Recall}}{(1 - \text{Weight}) \times \text{Precision} + \text{Weight} \times \text{Recall}} \right)$$

We have to define the fragment:

$$\text{fragment} = \frac{\text{Number of non-overlapping chunks in the candidate text}}{\text{Number of chunks in the candidate text}}$$

Such that we can define penalty as:

$$\text{Penalty} = (1 - \text{fragment})^{\beta}, \text{where } 0 \leq \beta \leq 1$$

And the weight as:

$$\text{Weight} = \frac{\text{Recall}}{\text{Precision} + \text{Recall} + \lambda \cdot \text{fragment}}$$

3.4.3 Hyper-parameter Tuning and Model Validation

Fine-tuning process for code summarization was most successfully ran using the hyperparameters in the figure 4. As it stands for bug-fixing, these values were changed throughout the research many times, however the values presented represent the best performing model.

Hyperparameter	**Value**
pretrained_model	microsoft/codebert-base
lang	java
source_length	256
target_length	128
batch_size	8
beam_size	10
train_batch_size	4
eval_batch_size	4
learning_rate	5e-5
epochs	10

Table 4: Fine-tuning hyperparameters code summarization

In order to obtain the presented hyperparameters, we had to try many different configurations of the values for each hyperparameter. This was mainly due to the limited computational and time capacity we had, and some parameters extended the training process for a long period of time. For this task we also employed the early stopping technique to avoid overfitting.

3.4.4 Early Stopping

For this task we implemented early stopping in a similar way to the bug-fixing as outlined in section 3.2.4. The only differentiating factor was the way we defined the intervals, when the evaluation and early stopping is considered. For code summarization, we had dealt with the different implementation of the fine-tuning algorithm that used epochs as way to train and signal progress. We adapted to this implementation by not setting a specific number of steps to run the evaluation after, but to run the evaluation 8 times per epoch. The value k was kept at $k = 5$ for this task as well.

4 Study Design

The objective of this particular study is to assess the viability of employing the CL procedure as a solution for training DL-based models that aid in automatically documenting code components, specifically Java methods, and automatic bug-fixing on the same language. The aim is to compare this approach with the conventional and widely accepted method of DL training. In particular, we answer the following research question.

4.1 Research Question

The research question we are considering is based on the change in metrics observed by the benchmark deep learning system, and the metrics observed by the employment of curriculum learning in the same deep learning system. Although the two tasks we are considering (bug-fixing and code summarization) are different in nature, we still propose a similar research question. This leads us to a generalized research question for this study that goes as follows:

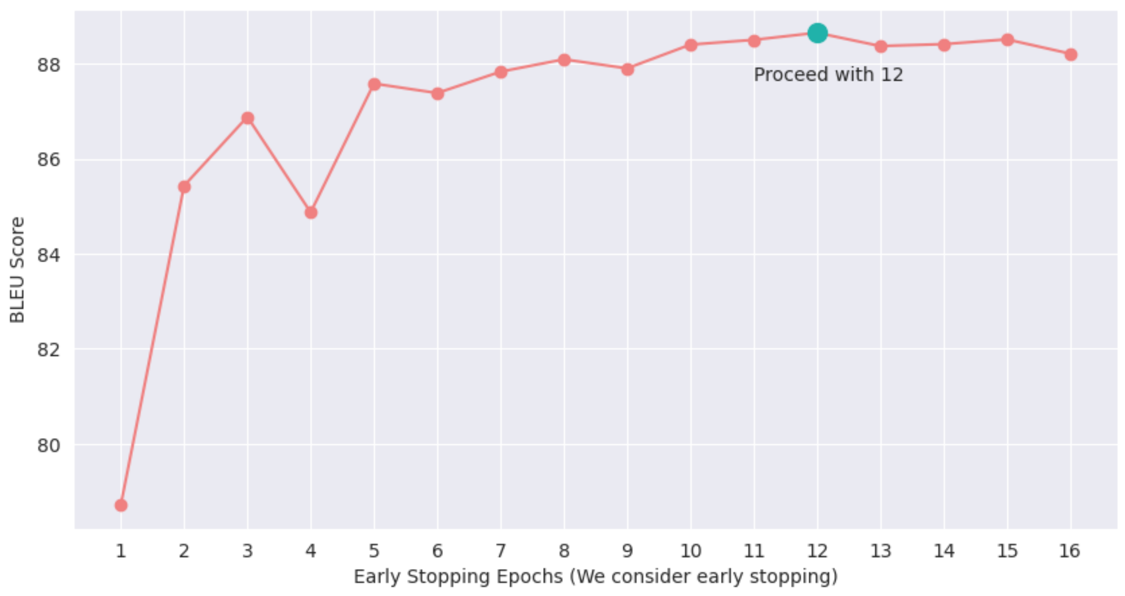
“ To what extent does curriculum learning influence performance of the deep learning code-based systems? “

More specifically this means we are interested in the performance improvements, if any, on both bug-fixing and code summarization when curriculum learning is applied.

4.2 Data Collection and Analysis

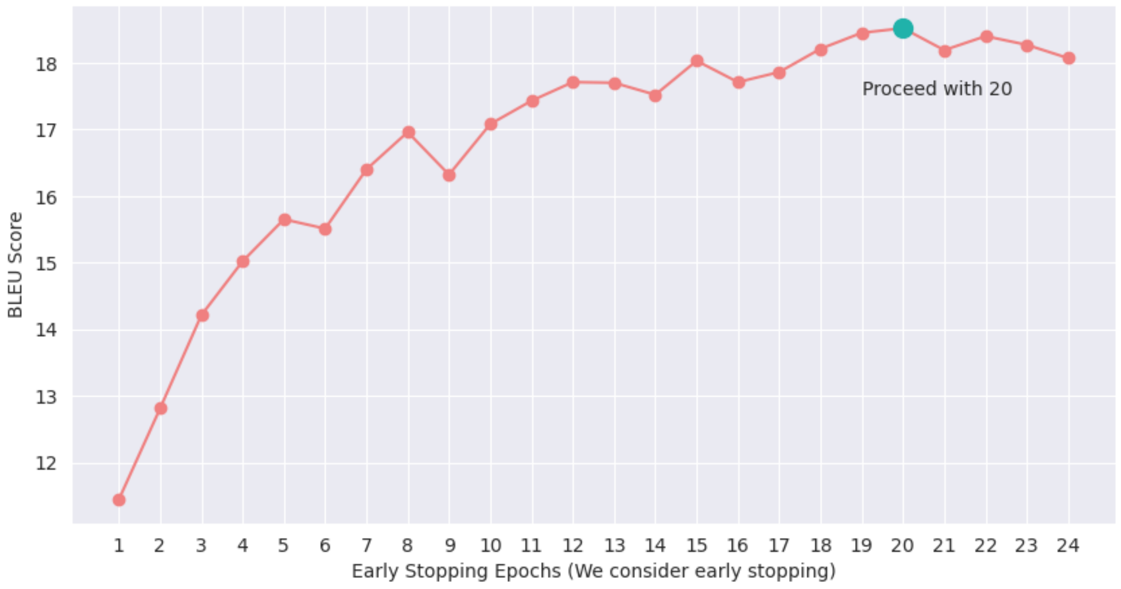
The manner in which training data was collected was outlined in 3.2.1 for bug-fixing and section 3.4.1 for code summarization. However, for obtaining the data later used to evaluate the performance of curriculum learning, we saved the output of each learning process to a corresponding *.log* file. CodeBERT produces a lot of output which was not valuable for this particular study, therefore we had to filter the irrelevant information and keep the BLEU scores. We used BLEU scores as a primary representation of a models performance, for a given time interval considering early stopping. We saved these values to analyze the learning performance and to identify the point at which the learning process plateaus. At such a point, we consider the checkpoint model created as final. The following figure 3 depicts an example of a learning process on the bug-fixing task, in this case with no curriculum learning applied.

Figure 3: Bug-fixing BLEU vs Early stopping epoch (Benchmark)



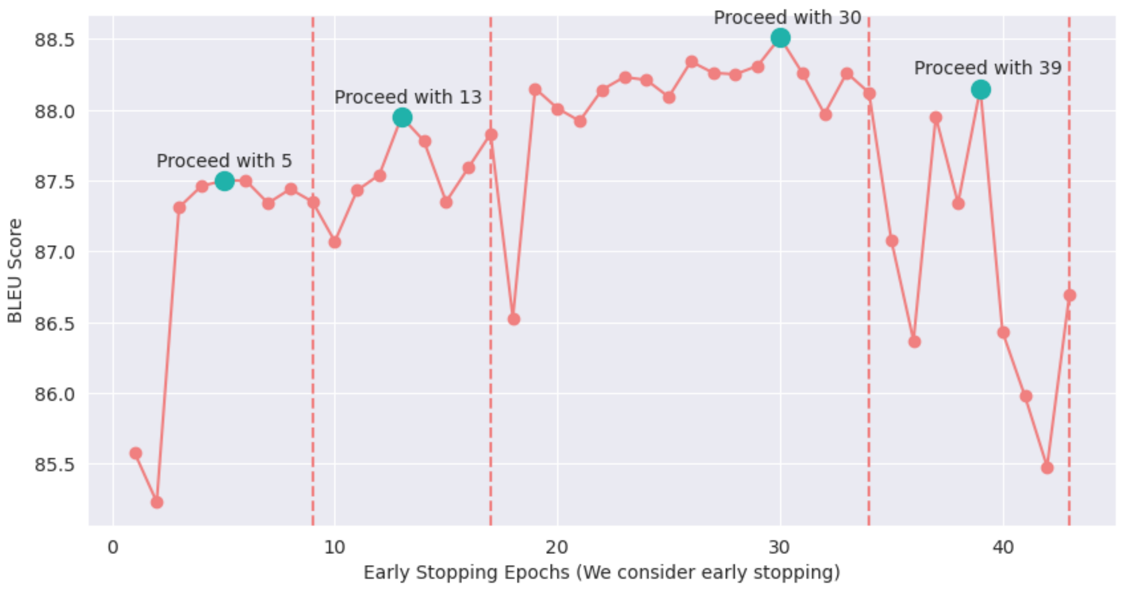
Notice the green checkpoint 12, the model created here is considered as the final trained model and used for inference. Same process was repeated for code sumarization yielding the following figure 4.

Figure 4: Code summarization BLEU vs Early stopping epoch (Benchmark)



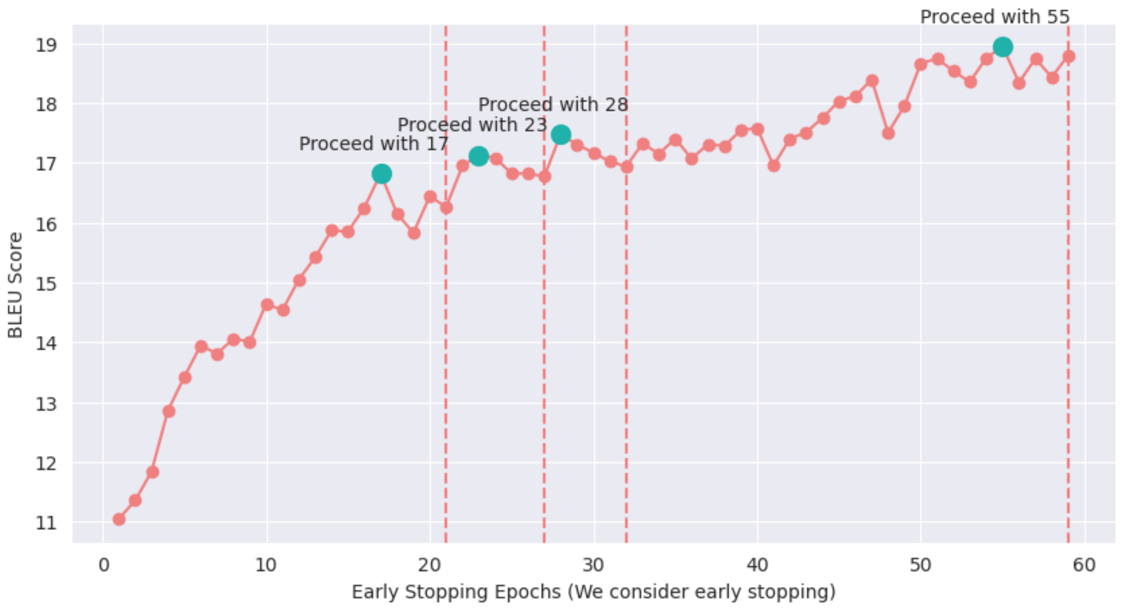
Considering we are applying the curriculum learning to the fine-tuning process. We would generate N plots for each individual batch we consider, while applying the same plateu (early-stopping) condition to halt the training. An example of a compiled version of all BLEU sores across all four batches in the context of the curriculum learning process on bug-fixing can be seen in the figure 5, where we identify each batch (restart of the fine-tuning process with higher class data) with a red vertical line. The selected checkpoint, indicated in green, for a given batch is used as a starting point (pre-trained model) to the consecutive step of curriculum learning.

Figure 5: Bug-fixing BLEU vs Early stopping epoch (Curriculum learning)



Same process was repeated for code summarization yielding the following figure 6.

Figure 6: Code summarization BLEU vs Early stopping epoch (Curriculum learning)



Now that we have the best performing models for the benchmark approach with no curriculum learning, and the best performing model of the last batch in the curriculum learned context, we can run inference on both with the hyperparameters as defined in section 3. Once the inference was ran, we obtained a table of metrics, for each configuration of the dataset (e.g. small, medium, small+medium), the hyperparameters and the learning methodology applied (no curriculum learning or curriculum learning). The table produced was the final indicator of the way in which changes in the learning process affect the evaluation metrics. The contents of the produced table will be further discussed in the section 5.

4.3 Replication Package

The experiment was conducted on the faculty cluster with relatively performant specifications. The CodeXGLUE project that was used to generate the results presented in this study can be found here, while the code and other assets created used during the study can be found here. The combination two repositories contains all necessary information to complete the fine-tuning process, with and without curriculum learning, as well as logs we obtained during our experimentation.

5 Results Discussion

As specified by in the *approach* section 3 and *data collection and analysis* section 4.2, we collected data from each fine-tuning attempt and recorded the logs produced by the fine-tuning algorithm. From these logs we extracted BLEU scores which were used as the early stopping criteria. Every time we fine-tune the model, we ran inefence right after which produced predictions. We used these predictions, together with the supervised test instances, to run the modified (with the metrics for this task) evaluation script provided in the CodeXGLUE project.

Considering we spent a significant portion of the research trying to improve the performance of the fine-tuning process, with and without curriculum learning, we have attempted to use multitude of combinations of hyperparameters, hoping for the right fit. In order to determine the hyperparameters which produce the model with the highest performance on the task-specific metrics, we saved the hyperparameters and the evaluation scores in a spreadsheet. From this spreadsheet we can determine the extent of the curriculum learnings performance contricutions in code-related tasks and reflect on the initial hypothesis.

As we proposed a general hypotheses which encompasses both tasks (bug-fixing and code summarization), we shall discuss the joint impact of curriculum learning on both tasks. Considering the *threats to validity* outlined in section 5.3 and the random nature of deep learning models, we can conclude that the curriculum learning on code-related tasks in the field of software engineering yields partially positive improvements in performance.

This is consistent with the idea of curriculum learning as a methodology of learning. While employing curriculum learning, we change only the ordering of the input of supervised instances which is not a substantial change in the fine-tuning process. This means that a partial improvement in performance is sufficient to deem the methodology as successful.

5.1 Bug-fixing

For bug-fixing we have tested 24 different hyperparameters and methodology configurations. This is due to the 2 learning methodologies (benchmark or no curriculum leaning and curriculum learning), 3 different datasets (small, medium and small+medium), 2 stopping criteria (early stopping and natural stop), 2 classification strategies (batch n contains batch $n-1$, and batch n contains only its own instances) and 3 different beam size hyperparameters used (1, 5, 10) for curriculum learning, while for benchmark only 1 was used. Once we had the table built with the previously mentioned hyperparameters and the metrics obtained for the inference predictions generated by the model of such hyperparameters, we can conclude on the performance of curriculum learning.

Figure 7: Results achieved for bug-fixing for both methodologies

Methodology	Bleu Score	Accuracy Score
Traditional Learning	88.41	6.58
Curriculum Learning	87.86	8.21

To generate the results in the figure 7 for both non-curriculum learning and curriculum learning, we used the following hyperparameters and methodology configurations: small+medium dataset, early stopping, CL batch contains instances of previous batch, bream size 1. As we can notice from the figure, the BLEU scores are similar with traditional learning taking the lead, while the Accuracy for curriculum learning is significantly higher. This indicates that the curriculum learned model is more accurate when evaluating the inference predictions and therefore should be more robust and reliable. To conclude this fully, more research needs to be conducted.

5.2 Code Summarization

For code summarization, we still used the same set of the hyperparameters as outlined in *Hyper-parameter Tuning and Model Validation* section 3.4.1. In total, for this task, we had 12 hyperparameters and the methodology configurations together with their scores in the table from which we extracted the followng figure 8.

Figure 8: Results achieved for code summarization for both methodologies

Methodology	Bleu Score	Rouge Score	Meteor Score	ExMatch Score
Traditional Learning	18.75	0.17	0.29	1.57
Curriculum Learning	18.26	0.17	0.28	1.15

As we can notice from the figure, the curriculum learning process did not help boost the performance of the fine-runing process. In fact the scores were lower then the benchmark for many metrics. That said it is difficult to conclude that the curriculum learning negatively impacts performance as changes in the ordering of the supervised instances does not directly interfere with the learning process. However, this might be an indication that while applying curriculum learning we might have overfitted on the parameters of the model, due to our classification strategy. The reason for the decrease in performance might also be caused by the nature of the task we are considering. It can be possible that generating natural language does not benefit from curriculum learning, although previous research suggests otherwise. In order to determine the reasoning behind the performance decrease for code summarization, more research needs to be conducted.

5.3 Threats to Validity

There are several factors that could have influenced this study to produce sub-optimal, or even erronious, results. Here, we list the most influential decisions made and constraints observed that could have impacted the study. We normally consider three different kind of threats: **(i.)** Construct, **(ii.)** Internal and **(iii.)** External.

5.3.1 Type (i.) - Inconsistencies in Metrics Used

It is clear that the more diverse set of metrics we use to evaluate the inference generated instances, the more holistic view we obtain on the performance and reliability of the underlying model. This is especially true considering that many metrics are explicitly used to reflect a specific property of a model. In our case we used two metrics for bug-fixing which was enough to conclude the impact of curriculum learning for this problem. However, for code summarization we decided to use four different metrics as we needed more insight into the behavior of the model prior to concluding on its performance. This begs the question, in hindsight, we could have also used more metrics to evaluate bug-fixing as some qualities of the model could have remained unmeasured with the two metrics used.

5.3.2 Type (ii.) - Batch Composition

Initially we contained instances of a given complexity range in a batch for that complexity range, and did not include simpler instances in this batch. This intuitively did not make sense as the fine-tuning system should, by the end of training, have all instances available at its disposal, and not just the last batch. We changed this and the composed the batches to contain the instances of the previous batch. This would mean that as we increase the complexity of supervised instances, we are re-injecting certain instances in multiple batches. More specifically we contain the simplest instances in all batches as they are contained in the first batch by complexity, and the following by the construction of the batch. Such a configuration have yielded better results, but this might have been a behaviour observed specifically on this dataset.

5.3.3 Type (ii.) - BLEU Score as Early Stopping Condition

During the fine-tuning process, we check the value of the BLEU score at a given evaluation cycle and compare it to the previous 5 in order to see if the score in question has leading performance. This means that we rely on a singular metric to employ curriculum learning, while the other metrics are only used at inference-time. This means that if a more complex early stopping condition was tailored specifically to evaluate the model during train-time, we could, most likely, achieve better performance, while ensuring less overfitting risk.

5.3.4 Type (iii.) - Faulty data or baseline project

There is a chance, although very small due to the peer review process, that the data and the paper on which we based this study is erroneous. This would imply that the results we achieved were also erroneous, as we did not conduct a detailed analysis of the underlying work or repeated the cited research.

6 Conclusions and Future Work

Due to the research-oriented nature of this project, we had to explore many heuristics which intuitively seemed promising, and test them against the system. This however, although time effective, was not heavily rooted in previous research. For the purpose of this study this was useful as novelty and exploration were key decision drivers. With the employment of a more robust complexity function and batching heuristic, we believe, the curriculum learning can be pushed to its limits, yielding improved performance over what was achieved in this study.

Prior to the study we decided to also explore the the following two tasks: Clone-Detection which is defined as an ability of the deep learning model to locate two code instance of same or similar functionality or syntax and Code-Translation which would specifically focus on translating Java code to C#. These, of course, were not studied due to time constraints and are tasks to be further explored.

The two tasks considered were fair representatives of code-related tasks and show a link to the application of a similar research for large software engineering code-based solutions. We propose further research to be conducted to evaluate the two tasks proposed, with the focus on the reasoning behind the failure of the code summarization to extend past the benchmark scores. it is important to note that the reasoning for this issue might not necesarily be the curriculum learning process itself, but a implementation errors or heuristic shortcomings.

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] What are recurrent neural networks? URL `https://www.ibm.com/topics/recurrent-neural-networks`.

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[6] University of San Diego Professional amp; Continuing Education. Curriculum design explained + 5 tips for educators, Nov 2022. URL `https://pce.sandiego.edu/curriculum-design-explained-5-tips-for-educators/`.

[7] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[8] Neel Kant. Recent advances in neural program synthesis. *arXiv preprint arXiv:1802.02353*, 2018.

[9] Stephanie Forrest. Genetic algorithms. *ACM computing surveys (CSUR)*, 28(1):77–80, 1996.

[10] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, 1992.

[11] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[12] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[13] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663, 2014.

[14] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi: 10.1145/3360585. URL `https://doi.org/10.1145/3360585`.

[15] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[16] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347. IEEE, 2021.

[17] Yuan Huang, Nan Jia, Hao-Jie Zhou, Xiang-Ping Chen, Zi-Bin Zheng, and Ming-Dong Tang. Learning human-written commit messages to document code changes. *Journal of Computer Science and Technology*, 35:1258–1277, 2020.

[18] Laura Moreno, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. Jsummarizer: An automatic generator of natural language summaries for java classes. pages 230–232, 05 2013. doi: 10.1109/ICPC.2013.6613855.

[19] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*, pages 200–210, 2018.

[20] Alexander M Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685*, 2015.

[21] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. An empirical study on code comment completion. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 159–170. IEEE, 2021.

[22] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[23] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[24] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[25] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55 (10):78–87, 2012.

[26] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[27] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[28] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.