Using Deep Learning to Generate Automated Code Documentation Across Multiple Languages

V N S Rama Krishna Pinnimty
Virginia Polytechnic Institute and State University
Blacksburg, Virginia, USA
ramapinnimty@vt.edu

Drew H Klaubert
Virginia Polytechnic Institute and State University
Blacksburg, Virginia, USA
kdrew17@vt.edu

ABSTRACT

In recent years, Deep Learning has been gaining a lot of traction in how it is capable of producing superior results given the models are provided with large enough training data. Many deep learning concepts have been around for a long time but the computing power needed to execute those algorithms has only recently broken through. With this breakthrough in computational power, deep learning has exploded in popularity and is being featured at the forefront of many new technological advancements. Researchers are now keen to develop deep learning algorithms to tackle issues that span across varied industries and disciplines. [6] The field of software engineering has taken an almost parallel trajectory in terms of its recent growth in popularity. This has prompted researchers to begin to study how deep learning and Software Engineering can be integrated with each other. The motivation for having the field of Software Engineering be aided by deep learning is to make the entire coding life-cycle easier for the programmer. What exactly could make this happen? The answer is simple. Anything that reduces the workload of the programmer and allows them to spend more time and energy on the task at hand would accomplish this. In this paper, we aim to present ways in which deep learning can be be used to assist in documenting code. In other words, given a code snippet, can we automatically generate comments so that the programmer can invest less time commenting code, and instead spend more time writing it? Consequently, achieving this will help answer our question of how deep learning can make the entire coding life-cycle a smooth process. In this paper we experimented to see which programming languages perform best with using models like CodeBERT for the use of generating natural language documentation given a piece of code. While we were unable to find a language that edged out the rest, our experiments are fruitful and speak volumes about the robustness of the models we are testing.

KEYWORDS

Deep Learning, Software Engineering, Code Documentation, BERT

ACM Reference Format:
V N S Rama Krishna Pinnimty and Drew H Klaubert. 2022. Using Deep Learning to Generate Automated Code Documentation Across Multiple Languages. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The explosion in the popularity of deep learning in recent years is hard for anyone in the technology industry to turn a blind eye to. Deep learning is helping power tons of the artificial intelligence that we see around us. There are many applications for deep learning but things like self driving cars, virtual assistants and natural language processing seem to stand at the forefront as mentioned in Deng [4]. One thing that all of these applications have in common is that they are aimed at making life easier for everyone. We can see from recent history and trends that the most popular applications for deep learning are the ones that are able to aid humans in some form of their everyday lives. This idea has recently found its way into the Software Engineering (SE) field. The integration of SE and deep learning is a very new and novel idea, but has a great potential for revolutionizing the industry and maybe, changing the way things are done forever.

Researchers thought of many SE-related applications which use deep learning to mainly assist the programmers in some way and also to make the overall software development process streamlined [7]. The one we will be focusing on is *code summarization.* Ahmad et al. [2] define code summarization as generating a readable summary that describes the functionality of a program. There are many reasons as to why one may need a summary for a piece of code. Reasons may include, aiding in the standard practice of commenting code to better work with collaborators, or even to use it as a guide to train new programmers and early developers to better comprehend code [13]. The reasons for needing code summarization can vary quite a bit but we will not be focusing too much on those. Our focus will mainly be to learn to use or create models for this task so that one could use our generated code summaries for whatever purpose they may need. Our overall task is to show that we can use pre-trained models in many different ways and that this process for training on new code can be streamlined and completed rather easily.

We are aware that there are many publicly available pre-trained models to accomplish this task [3, 10, 15]. Hence, we will be using those to their full extent. These models range from a simple Encoder-Decoder network, to more advanced and complex architectures like Transformer models, RoBERTa and CodeBERT. We will be attempting to use these models on data involving code and assess how we can fine-tune these models to suit our requirements. Our overall goal is to find out where these models succeed and struggle and put forth a detailed comparison and analysis taking inspiration from [18]. Also, we would like to report if certain programming

languages are more likely to have a more accurate generated summary than others. Should we be using different models for different languages or is there a single model that will work best across the board? These are the questions we would like to tackle and in doing so, we will be able to get a hands-on experience with the models and understand how they can be used in solving important real-world problems.

Contributions

As mentioned earlier, the two models we are trying to use to complete the code summarizing task are RoBERTa and CodeBERT. Along with a complete survey of these models and research on how they work, we will also be obtaining some results about their performance on codes from different languages. Hence the contributions that this work will provide are: -

- A comprehensive survey of the existing work that is done about the task of code summarization.
- A comprehensive overview of how each of the RoBERTa and CodeBERT models work.
- Experiments to show the performance of these models under the same hyper-parameter setting.
- Experiments to show the performance of these models on codes across multiple languages.
- A discussion about the results of the experiments along with their implications within the realm of software engineering.

2 RELATED WORK

To learn more about how the task of generating a brief summary for a code snippet has been done before, we will be taking a look at some of the related work. We are mainly looking to gain insight from this work regarding the experimentation that they are doing rather than the implementation of their models themselves due to the fact that we will be using pre-trained models from Wolf et al. [17] Hugging Face and fine-tuning them.

Hu1 et al. [8] developed a model for the code comment generation task using a Sequence-to-Sequence model (Seq2Seq), which will be one of the models that we attempt to use in our approach. Their work includes obtaining their own data to train the model with, whereas we will be using the pre-trained Seq2Seq model provided on the Hugging Face website due to time constraints of the class. According to their work, the Seq2Seq model is widely used for machine translation and text summarization. Their approach includes an encoder and a decoder which are both LSTMs and then an attention component. We really like the way their experiments were set up to test models against each other. We will definitely be taking inspiration from their experimental design but not necessarily using the same models.

Ahmad et al. [2] proposed a transformer-based approach for the task of code summarization. Their model architecture consists of stacked multi-head attention and parameterized linear transformation layers for both the encoder and the decoder. We learned slightly about this transformer architecture in lecture and its improvements over a baseline encoder-decoder model. This model's testing and experimentation was done with the same dataset that was obtained by Hu1 et al. [8] and a Python dataset that was used by Wan et al. [16] in their work which was aimed at improving automatic source code summarization using deep learning. We may find ourselves attempting to use these types of datasets to build on the one on CodeSearchNet.

LeClair et al. [11] introduce a way to improve the performance on the task of code summarization via a graph neural network. While we do not intend to use graph neural networks due to their complexity and our limited class time remaining, this piece of literature still proves to be helpful. Despite the graph neural networks being more complex, at their core, they do not seem to be too much different from the neural networks we have already studied. As a result, we can gain inspiration from their model architecture as well as their proposed plans and outcomes to be able to gauge if our goals are set at the right level for us and are an insightful addition to this particular field.

Liu et al. [12] presents the RoBERTa model and how it is an improvement on the existing pre-trained embedding algorithms such as BERT. This makes a few simple modifications to the BERT algorithm and they include (1) Training the model for a longer period of time using bigger batches accompanied by more data. (2) Removing the next sentence prediction objective, (3) training on longer sequences and (4) dynamically changing the masking pattern applied to training data. The paper gives an in-depth walk through of the BERT pre-training approach and then shifts the conversation to the RoBERTa model. We will be discussing this literature in much more detail in the following sections as this is the original paper that proposed the algorithm and the one that is most relevant to the pipeline that we are using from the CodeXGlue GitHub repository for fine tuning our model.

Feng1 et al. [5] Presents the CodeBERT model which is designed for allowing both natural language and programming language applications, one of which is code documentation generation which is the purpose of our project. CodeBERT was trained with a GitHub repository that contains code from 6 different programming languages. Two of these being Python and Java which is what we are focusing on in the scope of this project. This model is much like BERT and RoBERTa in that it uses a multi-layer bidirectional transformer architecture. This work is very important to us and much like the RoBERTa literature, we will be referring this work heavily as it gives us the information we need about the model that we are going to be fine tuning.

With these sources we feel that we have done a comprehensive literature review of the work related to our topic of creating natural language documents for pieces of code. Most of our literature was pertaining to different ways to complete the code comment generation task. We also have given an overview of the literature of the models that we plan to use to complete this task. Between the work that discusses different types of ways to complete the code comment generation task and the work that explains the models that we will explicitly be using, we have been able to learn a lot from the related work.

3 DATA/ MODEL DESCRIPTION

In this section we are going to discuss the data that we are using along with the model that we are implementing to complete our task. The data comes from the Husain et al. [9]'s CodeSearchNet dataset. This dataset was used for the CodeSearchNet challenge a few years back. The overall dataset contains 2 million entries, all of which are pairs containing code and their natural language comment. The comment in this data is a very top-level comment describing what the function is trying to accomplish. For instance, a java doc, if one was to be writing a function in java. On the other hand, the code portion contains the full function that the comment is describing. There are 6 different programming languages that are represented in this dataset all of which have different amounts of data associated with them. The languages that are present in the dataset are: Python, Java, JavaScript, Ruby, PHP and Go.

Recall that our hypothesis was that the Python code will perform better than the other languages because it is free-form and on the other hand languages like Java are more structured, as far as white space goes. For our purposes and to work within the time constraint of the semester, we will focus on only a handful of these languages. The ones that we will be using are: Python, Java, JavaScript and Ruby. To add to our hypothesis, we would like to make the observation that Ruby is a similar language to Python when it comes to structure. Java and JavaScript have a more formal structure needed in order to be syntactically correct. This means that we would like to pit Ruby and Python against Java and JavaScript to see which language gives the best performance.

Since these are the four languages that we are using, we are going to use the pre-processing pipeline given in the CodeXGlue GitHub repository to be able to extract only this data. We will not download the PHP and Go code. We will be running different models for each language to get a performance metric to be able to use for comparison. Since we will be running a lot of experiments and these models take a long time to run, we felt the need to truncate the data in order to be able to complete this within the time constraints of the semester. To truncate the data, based on our experience with experimenting with the data, we decided that 100,000 entries would be a sufficient maximum size for the dataset of each individual language to be able to have enough data but also be able to run the models in a timely manner. The only two languages that were affected by this truncation rule are Python and Java as Python had 251,820 rows and Java had 164,923 in their respective data sets.

Along with breaking down the dataset, we will also be doing a discussion about the models as well in this section. The primary model that we will be using is RoBERTa. We will also be doing some experiments with CodeBERT. In this section we will also give a brief overview of these two encoders.

3.1 RoBERTa

As mentioned in the Related Work section, RoBERTa is an improvement on existing language embedding algorithms like BERT, which use transformers. RoBERTa stands for Robustly Optimized BERT pre-training approach. This model is an improvement because it is trained for a longer period of time with bigger batches and more data. The original BERT model was trained for 1 million steps with a batch size of 256 sequences. This model was trained with 125 steps of 2,000 sequences and 31,000 steps with a batch size of 8,000 sequences. It is also trained on longer sequences. To reiterate, this is an improvement on the original BERT model and we will be adapting it to use on code for this project. While there is not much documentation on RoBERTa, there is plenty about the regular BERT model and it is rather easy to switch over to the new model with the BERT instructions. Figure 1 shows us the sample overview of the architecture of how RoBERTa works. We can observe that the architecture is very similar to the original BERT model.

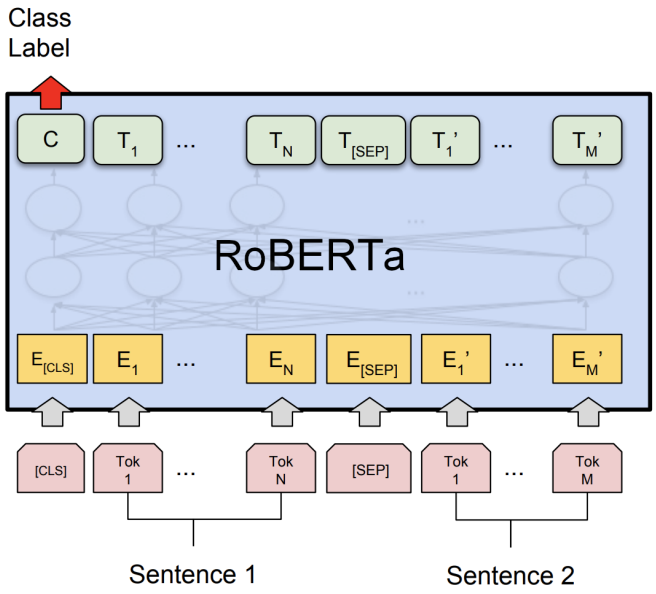


Figure 1: Simple architectural overview of the RoBERTa model; https://paperswithcode.com/lib/allennlp/roberta-common-sense-qa

Another way in which this is an improvement of the BERT model is because it removes the objective of next sentence prediction. This is something where the model is trained whether or not snippets of text come from the same document. The authors of the model were able to do some experimenting and noticed that the performance for the down stream tasks is slightly improved if this objective is removed from the pre-training process.

The architecture we are following for this project is an encoder-decoder architecture with tanh activation. The encoder for this process is the RoBERTa model and the decoder is the PyTorch Transformer decoder. This is the architecture that is used by the CodeXGlue repository given by Microsoft for the task of code-to-text which is the same as our task of finding the natural language comment for snippets of code.

3.2 CodeBERT

We will also do a little bit of an overview about CodeBERT. As we go through the experiments in this report, most, if not all of them will be done using the base CodeBERT model from Hugging face. We decided to do this because it seems to be the more state-of-the-art model at the moment as it tends to yield stronger evaluation

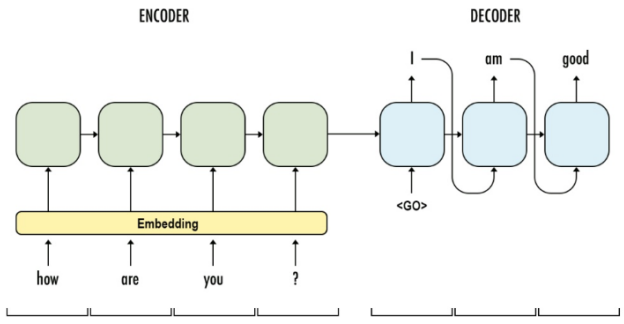


Figure 2: Encoder-Decoder structure of the RoBERTa model; https://medium.com/data-science-community-srm/understanding-encoders-decoders-with-attention-based-mechanism-c1eb7164c581

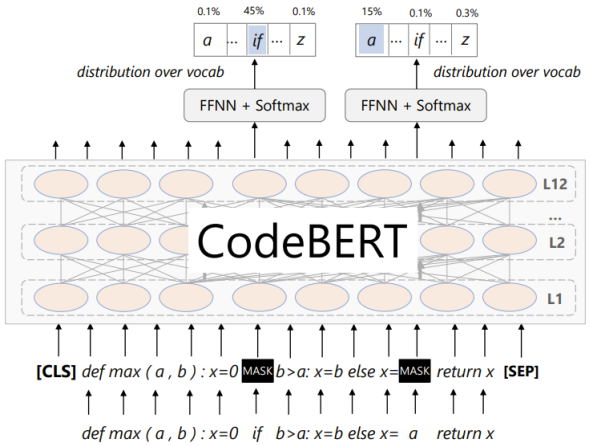


Figure 3: Simple overview for the general workflow of the CodeBERT model; https://bayesian-models.org/2021/08/17/kdd-workshop-on-programming-language-processing/

scores from the resources we have seen within the literature survey. As we mentioned in the related work section, CodeBERT is a bimodal pre-trained model for both natural language and programming languages. This model was trained on a dataset that was obtained from codes on GitHub repositories and it includes the same 6 programming languages as we previously mentioned. The bimodal data points mentioned in the training are pairs of code and their corresponding natural language documents. [5] The paper introducing CodeBERT states that training is conducted in a setting that is similar to the original BERT model. The development of CodeBERT is done using the same model architecture as RoBERTa which is part of the reason we decide to use RoBERTa for our task of comparing the results of codes since we can easily fine-tune it. CodeBERT was pre-trained with two objectives which include (1) Masked language modeling and (2) Replaced Token Detection.

Figure 3 shows us a simple diagram of the use of CodeBERT for a programming language (PL) task. The referenced literature also discusses that while the pre-training objectives of CodeBERT do not include generation-based objectives, they still explore the extent that CodeBERT can perform on these tasks using the CodeSearchNet Dataset. This investigation was done using fine-tuning of the model for a code to Natural Language (NL) task which is the same type of investigation we do in this project. The only difference being is their exploration included comparing models to each other, while we would like to compare programming languages together.

4 EXPERIMENT 1/ DISCUSSION

As we have and will mention, the inspiration for these experiments comes from the information in the CodeXGlue repository from Microsoft: GitHub Repository

4.1 First go around

In this section we are going to go through all of our experiments and discuss the results as we are able to when we present the output. There are quite a bit of experiments and hence we fine-tuned the model for the Code-to-NL task. Like we mentioned, these experiments will all be using the base CodeBERT model that is able to be retrieved from the Hugging face site. We mentioned it in the model section but the CodeBERT base model is what we would like to test because it was designed for tasks like this and we are looking to gain insights about the models behavior.

In previous sections, we have laid out our data a little bit and have stated that our goal is to compare the performance of the fine tuning of the RoBERTa model between different programming languages. To reiterate, the languages that we are going to be exploring are:

- Python
- Java
- JavaScript
- Ruby

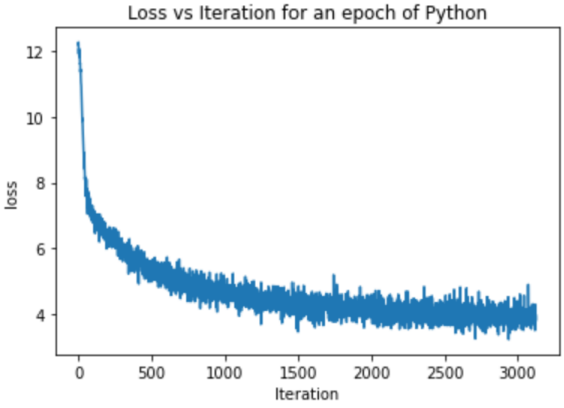
To evaluate the performance of the model for each of these tasks we will be using the BLEU score. This is generally the metric that we know more about and it is what we have used in the past so we will continue to use it as the metric for this project. It is also a very popular performance metric when it comes to encoder-decoder models and it seems it is standard practice to use it for this type of application.

As we also stated previously, we have broken the data up into partitions based on the language that the code comes from. So we essentially have 4 different datasets, all of which have codes from their own respective programming language. Since we have broken the data in this fashion we will also conduct our experiments in a similar way by going over the results of each language and what our takeaways are for that result. The model must be fined-tuned for each of the different experiments so doing this was a very time consuming process especially for larger amounts of data. In total, after all of our trials and analysis, we will have trained a total of 8 models i.e, two different experiments involving training the both the RoBERTa and CodeBERT models once for every language.

4.1.1 Python. In this subsection we are going to go over the experience regarding the Python data. Recall that since the Python data was very large, we had to truncate it in order to save time on training. We believe that so long as we do a similar amount of

fine-tuning on the model, each language has a fair chance of being able to show itself as the one that allows the model to perform better. The truncation of the data for the python codes took the amount of data from 251,820 to 100,000. We still believe this to be a sufficient amount of data.

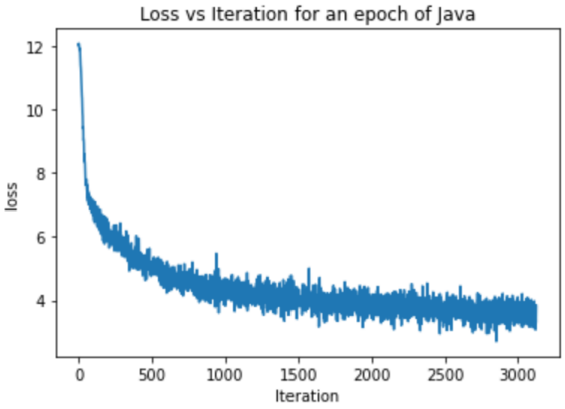
Using code that is inspired by the pipeline given in the Microsoft CodeXGLUE repository and the pre-processed data, we fine-tuned the model for the code-to-text task on the Python data. Below is the loss plot for that fine-tuning process.



Loss vs training steps plot of fine-tuning on the Python data for a single epoch

The loss plot above is the loss vs the batch updates for a single epoch of training on the Python data. Due to training times and the need to train many models for a comprehensive discussion of the results, we feel that a single epoch was enough training for us to draw a reasonable conclusion about our hypothesis. We can see that the model is training very nicely for the Python data and there does seem to be a bit of reasonable convergence even though it was just a single epoch worth of training. As we mentioned, the metric we are using is the BLUE score. The best BLEU score that we got for this data was a score of 15.43. To us this seemed rather high considering it was higher than most other scores reported by fine-tuning this model on GitHub. We also clearly did not train the model for nearly as long as it was trained compared to the 10 epochs that the original pipeline for fine-tuning gave it.

4.1.2 Java. In this subsection, we are going to go through the experiment regarding the Java data. We can also recall that this is another scenario where we needed to truncate the data down to something reasonable for us to train. Again the same rule applies. The original amount of data that was included that pertained to Java was a total of 164,923. This amount was brought back down to 100,000 just like in the Python experiment. To reiterate, the idea is that we do the same amount of training for each of the models and we will be able to draw reasonable conclusions. We once again used a single epoch of training. The amount of iterations still remains the same between the Java and the Python data because the amount of examples is 100,000 for each. The plot for the training iteration vs the loss is shown below.



Loss vs training steps plot of fine-tuning on the Java data for a single epoch

Just the same as before, we see that the model is training nicely and is able to get to some sort of convergence after the single epoch which is what we are looking for. We wanted to have just enough training where we saw a longer flat tail on the loss plot but also were able to train in a timely manner. So it it nice to see the plots behaving so well and showing us the exact result we are looking for.

We again need to consult the best BLEU score that we got for this model. For this model, we got a score of 16.0. Again this seems to be a bit high and so this confirms our suspicion that there may be something wrong with our hypothesis. We are not doing anything extraordinarily different from the guidance given by the fine-tuning noted from Microsoft. And to be clear, for this section of the data, we will still be able to draw a fruitful discussion about these results since the relativity to each other is more important than the numbers themselves from our perspective.

4.2 A quick break for discussion

We will take a break from presenting the results to debrief a little bit of what we have seen from the models so far. We are pleased with how nicely the models have trained and we have addressed the issue we seem to be having with the BLEU scores.

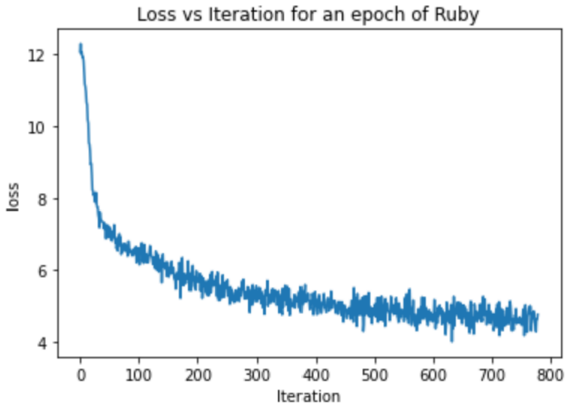
At this point in time, we do not have really have any evidence to support our hypothesis as the BLEU scores for both the Java and the Python trial are very similar. Recall our original hypothesis; we wanted to see if a language with more structure such as python would have been an easier language to tokenize and therefore have more accurate embeddings and down stream performance as a result of the better embeddings. So far if we have seen anything, we have seen the opposite result due to the higher BLEU score produced by the trial on the Java codes.

This is a very basic, surface level, introductory experiment with our hypothesis. We obviously need to do a lot more testing to see if we can come to a definitive solution. We will do another comparison experiment in the next section just like we have done. We will see a comparison between a more structured language and a less structured language. The two languages we are going to use for this are Ruby for the more "structured" language and then

JavaScript for the not so structured one. We will present the results of this experiment in an identical way as the previous one.

4.2.1 Ruby. In this subsection, we are going to go over the experiment regarding the Ruby data. In the previous experiments we found ourselves needing to truncate the dataset to be able to train within a reasonable amount of time but in this particular experiment, there is not quite as much data. The Ruby portion of the data set only contains about 24,927 training examples, so no truncation is necessary for us here. We do not have any issue with this amount of data, we still feel that we will be able to have enough training.

Again using the code that was aided by using the pipeline in the cited Microsoft repository, we fined tuned this model on the Ruby data for the code to text task. We show the loss plot vs the iteration number below.



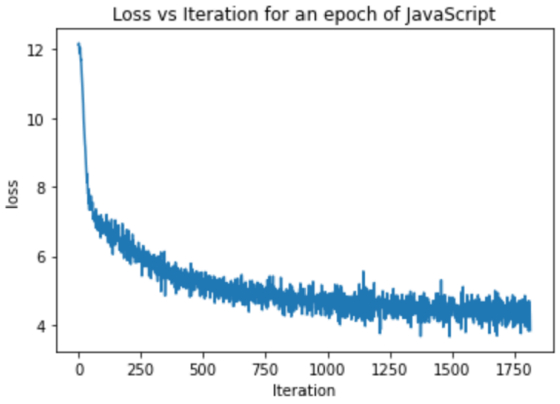
Loss vs training steps plot of fine-tuning on the Ruby data for a single epoch

As the other models have shown, the loss is reducing well and we are able to train the model to what looks like to be at least some relative convergence. It is important to note here that there are less iterations because of the lower amount of data that there was in the original set. The batch size of 32 has remained constant throughout the different models that have been trained, and we see no reason to change the batch size with the different amounts of data.

The best BLEU score that we got was a value of 10.32. We will hold off on making comments about this score until the discussion at the end of the section. We are still in the same scenario as the other languages and we at least have consistent behavior between the models. Let us look at the model that we get from fine tuning on the JavaScript portion of the dataset and see what we can conclude from there.

4.2.2 JavaScript. This experiment will round out the first section of experiments for our project. This experiment is going to fine tune the CodeBERT model on the JavaScript dataset. This is the less free form language that we are going to primarily compare with the results from the Ruby model. This JavaScript data was also a part of the data that did not need to be truncated because it was already small enough. The size of this original portion of data totaled to 58,025 examples.

For the final time in this initial experiment, we train the model. This time on the JavaScipt code-comment pairs. And once again we show the loss plot for this model just to make sure that it is actually learning something. This plot is shown below.



Loss vs training steps plot of fine-tuning on the JavaScript data for a single epoch

For the final time in this initial experiment, we see that the loss is decreasing as we would hope and we once again are able to train to a point of somewhat reasonable convergence. Again we must note that the number of training iterations is a little different for this because of the size of the dataset being a little different. We still used the batch size of 32. The BLEU score that we obtained from the testing on this model is 12.41. We will discuss the overall results in the next section.

4.3 Discussion of the initial results

In this section, we will be discussing the results from the different models in the experiment we just completed. First off, we are glad that the models seemed to train well. This was a tough model to get up and running for fine-tuning and training each of these took a very considerable amount of time. We would estimate that each of the models took a couple of hours to train for the single epoch. The smaller data sets obviously took a little less time just because there were fewer iterations. We present the scores that resulted from our models in Table 1.

Language	Training Size	BLEU Score
Python	100,000	15.43
Java	100,000	**16.0**
Ruby	24,927	10.32
JavaScript	58,025	12.41

Table 1: A table showing the results of the first experiment

From Table 1, we can see all of the BLEU scores as well as the training dataset size for each of the languages that we have trained the models for. As it relates to our hypothesis of languages like Python and Ruby having higher performance, we see no evidence that our hypothesis is true. In fact, if anything the opposite is true because Java seems to be the language that the model performed the best on. One thing that we need to explore more now is the effect of the size of the training sets.

It is apparent that the models that had more training data in them were able to perform better in the evaluation stage. This comes to

no surprise to us at all as this is a trivial finding for us at this point in the deep learning course. This is the reason why we needed all the datasets to be comparable in size to at least each other. This is why we tagged the Python and Java set together as well as the Ruby and the JavaScript sets together for our comparison study. However, in both cases we saw our hypothesis turn out to be wrong.

There is still more experimenting that can happen in this project. Now that we have hinted so far that our hypothesis is not correct, we would like to see if we can confirm it any further. In the next section we will run an experiment by training a series of models much like we did in this section. However this time, we will be making sure that the size of the training data is on an even playing field. The outline of that experiment is in the introduction of the next section.

5 EXPERIMENT 2/ DISCUSSION

In this portion of the project, we are going to do the same sort of experiment we just did in the previous section but we are going to cut down the total amount of data for each of the languages. The one issue we may have had in the previous section is that the size of the training data may be affecting the total scores more than we had originally been anticipating.

To do this, we are going to truncate all of the data and not just the data that pertains to Java and Python. In order to make it a fair fight for all of the models for the different languages, they are all going to be the same length. In order to do this, we must reduce the size of the data set for all the languages to be able to match the size of the smallest set. In our case this is the ruby data set which has only 24,927 training examples. What we finally decided on is to reduce all of the training sets down to an even 20,000 training examples. We are going to run through the same fine tuning process as in the previous experiment section to assess the efficacy of the tokenization and the embeddings on the different languages one final time. Once again the experiment sections will be broken up by language. Because we are drastically cutting down the size of the data for this experiment, we are able to run for more epochs. So for these models, we are going to training all of them for 3 epochs and then showing the loss for that.

5.1 Python

For the python section of this experiment, we will do almost exactly what we did for the previous part. We must recall that all of the data sets are going to be of size 20,000 for this and the following experiments. Which means we are going to train a new model for all of the languages. The good part about reducing the data to this size is that we have been able to cut down on training time just a little bit. The other experiments in the previous sections were all taking a couple hours to train with the exception of the Ruby data due to the already concise nature of the data. Now, the models are taking less than an hour to train which is very good considering we are still training multiple of them one at a time. The model that we have trained here is for the 20,000 entry Python data set. We are still using code that we have written that was inspired by the pipeline provided in the Microsoft CodeXGlue githib repository for fine tuning the model for the code document generation task. Recall that for this time around we are training for three epochs on the truncated data and we produce the plot of the loss vs iteration below:

Loss vs training steps plot of fine-tuning on the truncated Python data for three epochs

We can see from the plot above that we are getting a similar behavior to before where our model is training very nicely and we are getting a loss that is decreasing the way it should. One thing we feel we should note is that despite the fact that we are using significantly less data for this experiment in order to match the size of the smallest dataset comparison, we are still getting a plot that looks like the loss has almost converged. Obviously we could train for much longer until true convergence in order to get the best performance possible, but we needed to do it this way in order to make a fair assessment at the difference between the performance of the model on different languages.

The BLEU score that we are observing for this trial of the experiment is 14.15. We also just want to observe that this BLEU score is lower than the one in the first python model which makes a lot of sense because we obviously are training for much less iterations. Even three epochs of 20,000 training examples is less than the original data. Python was one of the better performing languages in the previous section so this will be good to figure out whether that could be attributed to the model working better with python or simply the fact that it was trained with more data.

5.2 Java

We will follow the same order of investigating the programming languages as in the previous section and move onto the Java data next. This was another one of the datasets that was very large and looking like it had good performance and we would like to figure out if that was due to the model itself or the fact that it had more data to be trained with. Recall in the previous section we actually saw better performance with the java data than the Python data.

We once again are going to train the model for the truncated data related to the Java language. The resulting loss plot for that training is shown in figure 4.

We make a similar observation here that there actually seems to be some reasonable convergence in the plot. We find this to be important because throughout the course we have been tasked with training model to convergence as opposed to having a set number of epochs for each of them. Whether this is because of a variation

Figure 4: Loss vs training steps plot of fine-tuning on the truncated Java data for three epochs

in computing power among the class or another reason, we have grown very akin to detecting convergence in these plots and are comfortable with the results we are getting with these models.

The BLEU score that we observe from the Java data comes out to 16.28. This is a very interesting result as it is consistent with the previous experiment in that it seems to be a little better than the python data. However we will debrief this result later on in the discussion part of this section.

5.3 Ruby

Much like before, the third chunk of data we are investigating for this experiment is the Ruby data. Recall that this was the data that had the fewest amount of training examples in the original data set. While this was the smallest set coming it at just over 24,000 training examples, we still truncated this a few thousand rows for the even 20,000 examples across the board. We could have easily just matched the size of this data set for all the rest but it was easier just to stick with a round number.

The loss plot for this run of training the model is shown below.

Loss vs training steps plot of fine-tuning on the truncated Ruby data for three epochs

Figure 5: Loss vs training steps plot of fine-tuning on the truncated JavaScript data for three epochs

We do not have much more to say that we have not already said about this plot. It is much easier to see the resemblance in the plots in this section of experiments due to the fact that they all run for the same amount of iterations.

The BLEU score that we observe in this model when testing it was 11.19. It comes to no surprise that this is not much different from the BLEU score in the last model that was trained on the Ruby data because the size of the dataset changed a very small amount so the model is not very different at all from the last time around. There is the exception that we are using three epochs instead of just one so the small performance improvement is expected.

5.4 JavaScript

The final language that we need to train the model with is JavaScript. This was a little like the Ruby model where we did not have to cut down the data in the first experiment but we had to cut it down in this one just by a little bit. The Plot showing the loss of the three training epochs for the JavaScript is shown in figure 5:

The BLEU score that we found with this model when testing it was 13.95. This is an interesting result as well and we will discuss it in the next section.

5.5 Experiment Discussion

We have concluded training the models for all of the different datasets for the different languages. Just like the first experiment, this is a total of 4 models. We have shown that they all are training nicely via showing the loss plot during training and we have also reported the BLEU scores for all of them. Those BLEU scores are consolidated and shown in table 2.

Language	Training Size	BLEU Score
Python	20,000	14.15
Java	20,000	**16.28**
Ruby	20,000	11.19
JavaScript	20,000	13.95

Table 2: A table showing the results of the second experiment where the size of the Data sets remains constant

To recap, once again our hypothesis was that the python and ruby models would perform better than the other due to the nature of the programming languages. We can see that the Python performance is the second best of the group an the Ruby performance is the fourth best of the group. This gives us no support for our original hypothesis. Since we cut down all of the data sets so the models would be training in an even playing field, we feel very confident in saying that our hypothesis is false. Between the two experiments we have trained eight total models and have done a complete survey through practical studies of the capabilities of the CodeBERT model when it is fined tuned for the code document generation task. We are not going to quite say that the opposite of our hypothesis is true in that the syntactically cumbersome languages are better than the ones with white space structure because JavaScript came in third place in the BLEU score evaluation. We feel comfortable saying that there is not much difference in the languages on the merit of the structure of the code which is what our experiment really aimed to get to the bottom of.

Our thoughts when it comes to this result is that CodeBERT is a very robust model and is able to make very good embeddings overall and does not rely on the qualities of any particular programming language. To us this is a very valuable result for researchers that are trying to study this model and the downstream tasks [14] that can be done with it. Something that someone could do in the future is to do this using the base RoBERTa model instead of the base CodeBERT model to see if fine tuning RoBERTa yields a different result. We would have done that ourselves but the nature of the testing of our hypothesis called for many models to be trained and we would have only had time to use one base model. So we decided on CodeBERT because it seems to be a little more state-of-the-art and has slightly better performance on downstream tasks. We will supply a python notebook that has the same experiment done but using the base RoBERTa model instead [1].

6 BROADER IMPACTS / DISCUSSION

With the results we have seen from our experiments, we can begin to discuss some broader impacts. Obviously we do not have results that were very exciting because the hypothesis that we stated did not turn out to be true. But we like that fact that we can still give valuable information from this result. For example we are able to know enough about the CodeBERT model to conclude that the way the model tokenizes the code or the text is very robust.

We had originally thought that since natural languages and programming languages have an inherent structure, the way we tokenize them could be different. For example, we usually think about tokenizing in the area of natural language processing (NLP) as splitting along the white space. Separating out each word where each word or punctuation is a token. If we extend that thinking to programming languages, we know that each language treats white space a little differently. For example we have been referring to Java as an structured type of programming language because it seems to not care much about white space as a delimiter for compiling code. Rather it uses the curly braces to do so. Java code can seem to look very messy and different between programs, even if those programs are doing the same thing. On the other hand when we think about Python code, we think about white space rules and structure. We think about the fact that the indentation matters for what areas of code are within certain loops or if statements. Additionally, Python also has a very natural language feel to it. For example using the word 'and' instead of the '&' as a way to say the 'and' conditional operator. The same goes for the 'or' as well as the negation of expressions. The natural language feel along with the importance of structure and white space led us to believe that the tokenization of these programs may be a lot more consistent and easy to predict and therefore may draw more accurate embeddings.

It turns out that was in fact not the case. This is an interesting result because at the end of the day it is a piece of positive news regarding the CodeBERT model. We feel that this means that the model is more robust than we had originally anticipated and therefore could be easier to train for other downstream tasks such as defect detection in code. This means that if we were to fine-tune the model to predict whether a code had an error in it or not, that we may not even have to train by the language like we did in this experiment, we can just train on a large set of codes with languages of all sorts included. This definitely makes fine-tuning a model like this a little more pleasant.

Overall, we feel that this piece of research has a positive impact. People who are getting into these models for the first time like our self when we were just starting the project, could learn a lot about how these models work on a granular level based on these experiments. We even hope that us having done this type of experiment can give motivation for new models that have not been developed yet. If new models that are in the process of development keep looking at themselves through the lens that we have looked at CodeBERT through during the process of these experiments, then that could make for more robust models. Now we are aware that this CodeBERT model is robust to our standards and did not necessarily need these experiments to be done on it but at least we have proven that this is the type of thought that should go into these models.

7 EFFECT OF THE REVIEWS ON OUR FINAL REPORT

We took the reviews from our proposal very seriously. Most of the reviews gave very helpful constructive criticism and feedback. Most of the feedback was directly related to our abstract section. We needed to make some mention of the hypothesis in the abstract which was not done in our proposal. We have made those changes accordingly, to adhere to the suggestions. We also were recommended to broaden the literature survey a little bit which we have also done. We believe our literature survey to be very comprehensive at this point. The final thing that we learned from the reviews is that we could have tied future work in the discussion a little more. We believe that we have made the trajectory of our experiments according to this idea. We like the fact that our experiments lead future researchers to have a direction in this domain if they want to continue the work. Overall, we really appreciate the feedback from the reviews. This was a great part of the proposal writing process and which definitely aided our writing of the final report.

8 CONCLUSION

To conclude this project report and subsequently the class, we would like to mention that we have learned a lot by doing this project. We have been able to get an extraordinary amount of hands-on experience with the CodeBERT model and show that we were able to use it. This experience is invaluable as we pursue the rest of our careers. Not only because we know something that is extremely useful in the field of data science right now, but we have shown that we are able to learn new things in the event that they are developed. Being able to adapt with the times is a skill that is paramount in the fields of software engineering and data science. We are glad to have had this experience and are thankful for this class for presenting us with these challenges. Learning to work with models like this is extremely challenging and we are very proud of the progress we have made in our learning and the things we were able to accomplish through this project.

9 WORK DISTRIBUTION

All of the necessary materials (code, datasets, etc.) from our experiments can be found by accessing our Google Drive repository:

https://drive.google.com/drive/folders/1ECC87H3V7n4m4Pxjm1T38vRJvRQ8zhX7?usp=sharing

9.1 V N S Rama Krishna Pinnimty

- Created jupyter notebooks for experiments
- Went through with the entire second experiment
- Slide development
- Proofreading paper/ preparation for final submission

9.2 Drew H Klaubert

- Initial experiment 1 - running the models and getting results
- Report writing and organization
- Slide development

REFERENCES

[1] Acheampong Francisca Adoma, Nunoo-Mensah Henry, and Wenyu Chen. 2020. Comparative analyses of bert, roberta, distilbert, and xlnet for text-based emotion recognition. In *2020 17th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*. IEEE, 117–121.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. (2020).

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[4] Li Deng. 2014. A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA transactions on Signal and Information Processing* 3 (2014).

[5] Zhangyin Feng1, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. (2020).

[6] Fabio Ferreira, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Software engineering meets deep learning: A literature review. *arXiv e-prints* (2019), arXiv–1909.

[7] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *2010 acm/ieee 32nd international conference on software engineering*, Vol. 2. IEEE, 223–226.

[8] Xing Hu1, Ge Li1, Xin Xia, David Lo, and Zhi Jin1. 2018. Deep Code Comment Generation. *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)* (2018).

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

[10] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. TreeBERT: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*. PMLR, 54–63.

[11] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. *IEEE/ACM 28th International Conference on Program Comprehension (ICPC)* (2020).

[12] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. (2019).

[13] Laura Moreno and Andrian Marcus. 2017. Automatic software summarization: the state of the art. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 511–512.

[14] Cong Pan, Minyan Lu, and Biao Xu. 2021. An empirical study on software defect prediction using codebert model. *Applied Sciences* 11, 11 (2021), 4793.

[15] Weizhen Qi, Yeyun Gong, Yu Yan, Can Xu, Bolun Yao, Bartuer Zhou, Biao Cheng, Daxin Jiang, Jiusheng Chen, Ruofei Zhang, et al. 2021. ProphetNet-x: large-scale pre-training models for English, Chinese, multi-lingual, dialog, and code generation. *arXiv preprint arXiv:2104.08006* (2021).

[16] Yao Wan, Zhou Zhao, Min Yang, Guandong, Xu, Haochao Ying, Jian W, and Philip S Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018), 397–407.

[17] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[18] Yuxiang Zhu and Minxue Pan. 2019. Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352* (2019).