Survey of Embedding Performance for Semantic Code Search

Shishir Jakati **Sean Kelley** **Mukul Kudlugi** **Sidharth Subramanian**
UMass Amherst
{sjakati, sgkelley, mkudlugi, sidharthsubr}@umass.edu

1 Problem statement

Semantic search has been a critical area of research with in the field of Natural Language Processing (NLP) since its formal inception. At its core, semantic search seeks to improve the accuracy of existing lexical searches by utilizing a deeper, contextual understanding of the query in order to provide more applicable and actionable results. Recent advances in the the related field of word embeddings shows promise in providing the foundation for the contextual understanding needed for semantic search.

Despite these developments, semantic code search, a subfield of semantic search, has seen very little direct or indirect progress. Semantic code search involves the task of using a natural language query to retrieve code snippets that perform the desired function (Husain and Wu, 2018). For example, a programmer might search ”join array elements with commas,” and a search engine would provide the following Python code: `','.join(arr)`. Unlike previous domains where semantic search has been applied, semantic code search presents a unique set of problems inherent to programming. The tokens used to write code are often technical and highly specific to their respective codebase. Likewise, natural language queries for code contain domain-specific terminology and occupy a vast range of specificity.

To further progress current performance on semantic code search, GitHub announced the CodeSearchNet challenge (Husain et al., 2019). The team collected a dataset of functions written in Go, Java, JavaScript, PHP, Python, and Ruby, some with accompanying documentation. The dataset contains six million total functions, with two million that have documentation. In addition to collecting the data, the team has also created evaluation metrics and a public leaderboard to track the current state-of-the-art.

In their paper introducing the CodeSearchNet challenge, the GitHub team defined their baseline neural model architecture which relies heavily on embeddings. During training, each query and code snippet are encoded into a shared vector space with the objective of minimizing the distance between query vectors and their corresponding code snippet. Thus, the backbone to this approach is to learn an optimal embedding representation.

In our work, we focused on exploring the efficacy of different word embedding methods when integrated into the existing GitHub model architecture. Each method was chosen to have a direct comparison to a baseline model and therefore make performance comparisons more meaningful. We hope that our results can be used to establish new baselines and further the latest research in this domain.

2 What was proposed vs. what was accomplished

- ~~Understand codebase~~
- ~~Setup training environment~~
- ~~Implement Word2Vec~~
- *Implement RNN Attention*

  Our Attention implementation was abandoned due to repeated Tensorflow errors, which we encountered specifically when transposing the input stream of tokens. Peculiarly, we were able to train our network for a significant number of iterations before encountering these errors. In future works, it would be important to revisit these errors.

- *Implement ELMo*

  Working with the publicly available codebase, we attempted to implement the ELMo model as an encoder, but we ultimately abandoned the approach since we were not able to integrate the stream of tokens with out ELMo encoder. In future works, we would look to rearchitect the way in which the stream of tokens is ingested.

- ~~Implement Concat Pooling~~
- ~~Implement Concat Pooling + Learned Weighted Avg~~
- ~~Evaluate performance of all implemented models~~
- ~~Error Analysis of model results~~

3 Related work

3.1 Word Embeddings

The development of optimal embeddings has been and continues to be possibly the most active area of research in NLP. The classic approach, known as bag-of-words, represents each type in the vocabulary with a one-hot vector. Although simple and intuitive, this method fails to encode the meaning of the words or any positional information.

Mikolov et al. (2013) sought to address some of these drawbacks with the creation of Word2Vec. In short, real-value vectors were learned such that words that appear in shared contexts would be closer to each other than other unrelated words. This representation has been shown to capture relevant information about the meaning and usage of words (Mikolov et al., 2013; Pennington et al., 2014) and better downstream performance (Collobert et al., 2011), however, this representation fails to capture any data-specific contextual information about each word that is encoded.

More recent techniques have sought to inject context into word embeddings by extracting representations from language models. In the most basic of these techniques, the last hidden layer of a recurrent neural network (RNN) is extracted as the embedding. More sophisticated methods combine multiple layers of bidirectional language models (Peters et al., 2017, 2018) or utilize self-attention in transformer architectures (Devlin et al., 2018).

3.2 Semantic Code Search

Previous work was done by the GitHub engineering team in this blog post (Husain and Wu, 2018). Their approach involved training a Seq2Seq model to act as an encoder for the code and training a separate language model to use as a sentence encoder for the queries. The code embeddings were then mapped into a shared vector space with the sentence embeddings using dense layers. Finally, a similarity lookup was performed to match an input query with a function.

More recently, Cambronero et al. (2019) investigates the overall efficacy of supervised learning techniques on the specific problem of semantic code search. They accomplish this in part by comparing an existing unsupervised method known as Neural Code Search (NCS) (Sachdev et al., 2018) with their own approach, Embedding Unification (UNIF), which is effectively a supervised NCS. Although it is unclear whether better performance is achieved with these methods, the paper offers a critique of the usage of documentation as a surrogate from natural language queries.

3.3 Code Summarization

The problem of semantic code searching must rely heavily on the embeddings, and subsequent encodings, the underlying models produce. For this reason, it is important that any deployed model is able to summarize code in a way that is then referable. (Allamanis et al., 2016) approaches the problem of code summarization through neural attention mechanisms. In using attention mechanisms, they are able to condense short code snippets into shorter phrases that summarize the functionality of the code. This work is significant since it demonstrates the ability of a neural method to understand important context-dependent features of the original inputted code snippets (Allamanis et al., 2016).

4 Our Dataset

The provided dataset is publically available and extensively documented at https://github.com/github/CodeSearchNet. It is segmented into various partitions by programming language, and subsequently into training/testing/validation splits. Specifically, the data is formatted into JSONL files, newline-delimited JSON files. Additionally, the dataset sequesters those functions that do not have an associated docstring as additional data points. Table 1 is replicated from Husain et al. (2019) which shows the distribution of the available data across both language and corresponding documentation.

A single datum follows the following schema:

- repo: the owner/repo
- path: the full path to the original file
- func_name: the function or method name original_string: the raw string before tokenization or parsing
- language: the programming language
- code: the part of the original_string that is code
- code_tokens: tokenized version of code
- docstring: the top-level comment or docstring, if it exists in the original string
- docstring_tokens: tokenized version of docstring
- partition: a flag indicating what partition this datum belongs to of train, valid, test, etc. This is not used by the model. Instead we rely on directory structure to denote the partition of the data.
- url: the url for the code snippet including the line numbers

Due to the format of this document, it would be difficult to accurately display an example data from this dataset and thus we provide this link for a python notebook which displays an example and some relevant statistics. As would be expected, the tokens generated for the code snippets are vastly different from even a byte-pair encoding of natural language. We expect that the unusual types of tokens make transfer learning using models pre-trained on natural language like BERT much less effective. In addition, many provided docstrings are quite small and describe the functions at very high levels which could make learning representative embeddings difficult. Unfortunately, some programming languages have much less viable data and so we expect model performance on held out data from those languages to be worse.

	Number of Functions	
	w/ documentation	All
Go	347 789	726 768
Java	542 991	1 569 889
JavaScript	157 988	1 857 835
PHP	717 313	977 821
Python	503 502	1 156 085
Ruby	57 393	164 048
All	2 326 976	6 452 446

Table 1: Dataset Size Statistics

5 Baselines

The GitHub team provides baselines using a joint vector representation (JVR) as well as ElasticSearch. In the context of semantic search, a JVR can be learned such that the vectors of encoded queries and corresponding results will be ”close” to each other in some latent space. For this project specifically, this architecture would encode some natural language query and then return a list of code snippets that, when also encoded, were closest to the query. ElasticSearch, unlike JVR, is not a neural model and is instead based on the Apache Lucene project. Since we focused entirely on investigating the performance of additional encoders for creating JVRs, we will not be including specific information about the ElasticSearch baseline here.

The following encoders for JVR are available as baselines: Neural-Bag-Of-Words, RNN, 1D-CNN, Self-Attention (BERT), and a 1D-CNN+Self-Attention hybrid. Through the Weights & Biases integration, we have access to the corresponding model weights and statistics. Below we have included a short description of each baseline model. Note that the hyperparameters for all of the models are presented in Table 7.

5.1 Encoders

Neural-Bag-Of-Words (NBoW) The Neural-Bag-Of-Words technique simply takes the traditional Bag-Of-Words method and makes it differentiable such that it can be used in neural models.

RNN Recurrent Neural Networks (RNN) augment the normal neural network structure to better retain information from previous inputs. This is especially useful when sequences are used as input. In short, an additional weight matrix is learned which controls how much information from the previous hidden state is included in the current hidden state.

1D-CNN Convolutional Neural Networks (CNN) are designed to use a set of filters or kernels to learn spatial hierarchies in input sequences. In the 2D case of images, CNNs can learn increasingly higher level features, from lines and curves to the complex structure of the target in the image. In the case of text input, a 1D filter is used but the underlying architecture remains the same.

Self-Attention (BERT) Bidirectional Encoder Representations from Transformers (BERT) was the first truly bidirectional encoding method for text. It accomplished this by masking at most 15% of the input sequence. The transformer then performs the task of a language model and predicts the masked tokens. For improved accuracy, the model also predicts whether two input sequences appeared sequentially in the data.

1D-CNN+Self-Attention This model combines the aforementioned architectures into one pipeline. There does not seem to be any research which directly examines this method but it was included by the GitHub team.

5.2 Note on Sequence Embeddings

In their baselines, the Github team has implemented three basic methods of sequence embedding, all involving pooling. Specifically, they implements global max, mean, and learned weighted mean pooling over the token sequences.

5.3 Training Data Setup

In accordance with the GitHub team, we utilize a 80/10/10 train/val/test split for our experiments. To further ensure our results are consistent, we did not modify included the code which prepares this split.

6 Your approach

Given our intention to provide a survey of embedding techniques in this domain, for more relevant comparison we chose each of our techniques such that they could be compared to a similar baseline and that we could successfully implement and test them in the time allotted. In each subsection below, for completed models, we summarize the model architecture, which baseline we intended to compare it to, implementation challenges, its location in the codebase, and the results using the provided evaluation metrics. For models we were not able to complete, we detail the challenges that we could not overcome.

6.1 Token Embedding

The method by which tokens in a corpus are encoded continues to be the fundamental problem in NLP. For our survey, we chose to include a variety of methods from recent research on embedding techniques to supplement the presented baselines.

6.1.1 ELMo Encoding

Tensorflow Hub (Abadi et al., 2015) provides an implementation of the ELMo (Peters et al., 2018) architecture, complete with pre-trained weights. We believed this model to be a viable choice of encoding both our queries and code due to the fact that encodings would be contextual. As we have mentioned above, functions in our dataset are implemented in a varied selection of languages that all have separate syntax and control structures. For this reason, we believed a the explicit contextual nature of the ELMo encodings would produce a performant encoder. We can see the code implemented at this pull request.

Ultimately we abandoned the approach due to the fact the publicly available codebase did not provide efficient access to the raw tokens. Specifically, the encoder module makes the assumption that the core model would operate on embeddings produced by a linear embedding layer. We attempted to overcome this by referencing the token vocab dictionary, but this did not allow for an efficient implementation. Additionally, we attempted to use the raw tokens from the original generator function, but ran into issues with tensorizing these tokens.

6.1.2 RNN Attention

This model will emulate the Attention model we worked on for one of the homeworks in the class. A major challenge was that we had to adopt the homework implementation to the GitHub codebase. Another challenge we faced was the fact that the codebase uses Tensorflow instead of Pytorch. While they both can accomplish the same functionality, we spent a considerable amount of time experimenting and reading the docs to make sure

	Encoder			CodeSearchNet Corpus (MRR)						
	Text	Code	Pool	Go	Java	JS	PHP	Python	Ruby	Avg
Github	NBoW	NBoW	LWA	0.6409	0.5140	0.4607	0.4835	0.5809	0.4285	0.6167
	1D-CNN	1D-CNN	LWA	0.6274	0.5270	0.3523	0.5294	0.5708	0.2450	0.6206
	biRNN	biRNN	LWA	0.4524	0.2865	0.1530	0.2512	0.3213	0.0835	0.4262
	SelfAtt	SelfAtt	LWA	0.6809	**0.5866**	0.4506	**0.6011**	**0.6922**	0.3651	**0.7011**
Ours	NBoW	NBoW	CP	0.4949	0.2242	0.174	0.0801	0.1367	0.2858	0.2497
	NBoW	NBoW	CP+LWA	0.5846	0.3926	0.3333	0.2928	0.3867	0.3732	0.4495
	biRNN	biRNN	CP	0.4630	0.2933	0.0913	0.3589	0.2985	0.0856	0.4357
	CBoW	CBoW	LWA	**0.6903**	0.5483	**0.4826**	0.5416	0.6492	**0.4554**	0.6600

Table 2: Embedding performance comparison between GitHub's baselines and our models using Mean Reciprocal Rank (MRR) on Test Set of CodeSearchNet Corpus. The "Pool" column indicates the pooling method used on both the text and code encoders: Learned Weighted Average (LWA), Concat Pooling (CP), and both (CP+LWA). Note, we corrected an error from Milestone 2 with the Avg value for our NBoW with CP encoder.

	Encoder			CodeSearchNet Challenge– NDCG All						
	Text	Code	Pool	Go	Java	JS	PHP	Python	Ruby	Avg
Github	ElasticSearch		-	**0.186**	**0.190**	**0.204**	**0.199**	**0.256**	0.197	**0.205**
	NBoW	NBoW	LWA	0.130	0.121	0.175	0.123	0.223	**0.212**	0.164
	1D-CNN	1D-CNN	LWA	0.059	0.128	0.044	0.135	0.166	0.115	0.108
	biRNN	biRNN	LWA	0.019	0.062	0.025	0.045	0.064	0.030	0.041
	SelfAtt	SelfAtt	LWA	0.049	0.100	0.061	0.094	0.181	0.190	0.113
Ours	CBoW	CBoW	LWA	0.104	0.139	0.15	0.08622	0.1727	0.2017	0.1423

Table 3: Normalized Discounted Cumulative Gain (NDCG) Results on CodeSearchNet Challenge leaderboard.

the operations that were being carried out were the same.

The Attention RNN was going to be compared with the RNN from GitHub as our baseline. The idea was to see if implementing attention would beat their RNN. If it doesn't, then it probably won't do any better than their RNN when compared to other models. If Attention beats their RNN, then we could further compare it with other models to see how improved the results are.

As mentioned earlier, we implemented attention the same way we did for the homework assignment. Due to the issues mentioned above, we spent a lot of time understanding the nuances of TensorFlow and its methods and how it works as we developed the model. Furthermore, we had to write the model to suit the codebase we had. In other words, we had to make our code fit with the template provided to us by GitHub. This took quite a bit of time, since we also had to discern how much of the processing the provided code did for us. This was important to know, since this would inform exactly what parts of attention needed to be implemented. The code that was completed can be viewed in this pull request.

We attempted to implement attention within the RNN sequence encoder. In the homework assignment, we encoded the data using an RNN, performed Attention on this encoded data and concatenated context vectors. The result of this operation was then passed into the decoder. We followed a similar approach, where we inserted our attention code into the existing encoder itself, using the outputs from their RNN implementation. After the context vectors are concatenated to the final states, the token embeddings are pooled and the resulting sequence embeddings returned. To enable attention, we added a hyperparameter called "rnn_do_attention", where if it is true, our added code would execute, otherwise, the RNN sequence encoder would run just as it would have before we added Attention.

Unfortunately, we could not get Attention to work as we intended. In our implementation of Attention, we had to perform a lot of transposing. This required knowing the dimensions of the Tensors being passed through. However, after running the code, we found that some Tensors had different shapes, which could not be accounted for. Therefore, the results for this model have not been found.

6.1.3 Continuous Bag of Words (Word2Vec)

Continuous Bag of Words (CBoW) is one of two different Word2Vec models, Skip-gram being the other. CBoW predicts the current word given its surrounding words. For example, the sentence ”He likes pie” would use [He, pie] as the target window to predict ”likes”. The model works by first creating a context or target windows for each word in the corpus. Then, given a context, the model tries to predict the target word by classifying the context in a neural network.

Our CBoW results will be compared to Github’s NBoW baseline, as both models are similar in implementation. Considering the NBoW implementation was the best neural baseline provided by GitHub, it stands to reason that CBoW will also perform well.

For our implementation, we decided to use a target window of size 4, with 2 words on each side of the current word. If there weren’t 2 words on one side of the current word, we used as many words we could on that side and the missing words were input as zero vector token embeddings. The Continuous Bag of Words implementation can be seen in this pull request.

As seen in Table 6, the MRR for CBoW performs significantly better than NBoW in every individual language and in average MRR. CBoW also beats Self-Attention on some languages. In their paper, the GitHub team mention that the Self-Attention model’s high scores was due its impressively large capacity. Our model, CBoW, only has slightly more capacity than NBoW relative to Self-Attention and thus we suspect that there is a upper limit where capacity, especially in the domain of token embedding, becomes superfluous.

As we further explain in the error analysis, we intended to submit our best model to the leaderboard in order to obtain the NDCG scores. CBoW was our best model as measured by MRR and thus its predictions were submitted. Our public submission can be found at this link. Although we were able to achieve 9^{th} place on the leaderboard unofficially, we expected that CBoW’s vastly improved performance in MRR compared to the NBoW baseline would be indicative of a NDCG score higher than NBoW. However, as clearly visible in Table 3, CBoW only outperformed the NBoW baseline on one language. This result is quite interesting and we explore some possible implications in the error analysis.

6.2 Sequence Embedding

In order to allow a semantic search between natural language queries and corresponding code, a strategy must be employed that effectively utilizes token embeddings from both spaces to compute or learn embeddings for entire sequences. Inspired the GitHub team’s first blog post on semantic code search, we decided to implement concat pooling to compare to the learned weighted mean pooling used by the baseline models.

6.2.1 Concat Pooling

Concat pooling, as proposed in Howard and Ruder (2018), concatenates the max and mean pool of the hidden states ”over as many time steps as fit in GPU memory” to the final hidden state. Howard and Ruder present this method as a technique for fine-tuning the ULMFiT method for ”Target task classifier fine-tuning”. The motivation behind this technique was that the last layer of their AWD-LSTM (Merity et al., 2017) would likely not be sufficient for classification of a sequence or entire document.

Initially, we thought that this alternative sequence embedding method could be compared to all of the provided baselines as they were trained using a basic learned weighted average. However, as briefly explored in our second milestone, it became clear that, unless we devised a novel modification of this method, it only made sense to use concat pooling on recurrent models, as originally presented by Howard and Ruder (2018). Thus we will only be comparing the unaltered implementation to the biRNN baseline. However, we will more deeply describe our change in direction as well as an attempt at a novel modification using a combination of concat pooling and the learned weighted average in the error analysis.

Within our fork of the CodeSearchNet codebase, we have added a new pooling option, `'concat'`, which will perform the operation described above using the max and mean pool functions already implemented by Github. Most of the repository operates on the assumption that the user has correctly set the encoder hyperparameters such that there is no dimension overflow when creating the embeddings. We originally thought that this assumption would be sufficient to incorporate the new pooling method. However, none the pooling methods implemented by the GitHub team changed modified any dimensions and thus the the sequence embedding dimension was equivalent to the token embedding dimension. In addition, the codebase is designed such that a TensorFlow built-in embedding layer is applied before the encoders are initialized, there was no way to set the hyper-parameters of certain models like NBoW to prevent overflow. Therefore, we have added the hyperparameter `'seq_embedding_size'` to all encoders that allows the sequence embeddings size to be set independently of the token embedding size. Now, when concat pooling is used, if the default token embedding size of 128 is used, the sequence embedding size should be set to 384.

The pooling method was added to src/utils/tfutils.py along with small changes to other related files. The entire diff can be viewed in the following two pull requests: 1 and 2. While testing our implementation, we ran into a considerable amount of errors and we created an issue in the repository to keep track of our progress.

Tables 4 and 5 present the MRR results of all the encoders that we trained with concat pooling or a variant. Both NBoW with normal concat pooling and concat pooling with a learned weighted average perform worse than the NBoW baseline on every language. Although clear in hindsight that the NBoW encoder with concat pooling would not perform as well as the original baseline, it is not clear why the other also performs worse. The biRNN encoder with concat pooling manages to out perform the biRNN baseline on four out of the six languages as well as the average. This result is more in line with what we expected. All these results and their implications will be further detailed in our error analysis.

6.3 Training Environment

As recommended by the GitHub team, we made use of the Amazon Web Services (AWS) EC2 p2.xlarge and p3.2xlarge instance types. Although we previously thought there would be an issue with credit usage for training, it appears that the AWS Cost Explorer is completely inaccurate and we did not spend more than our $350 limit. Each instance used the Deep Learning AMI (Ubuntu 18.04) Version 25.0.

Additionally, we loaded Docker onto our instance so that we could utilize the prepared containers, as provided by Github. These containers make use of the NVIDIA Docker runtime which provides GPU support. Once the containers are loaded, we were able to download and store the preprocessed datasets, which are also provided by Github. Further details can be found on the official README.

7 Error analysis

To evaluate the performance of their models, the GitHub team uses two methods different methods: Mean Reciprocal Rank (MRR) and Normalized Discounted Cumulative Gain (NDCG).

MRR is defined as follows

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_i} \qquad (1)$$

where Q is a set of queries and rank_i is the position of the first relevant result for the i-th query. In the case of the CodeSearchNet paper, the team selected a fixed set of 999 ”distractor snippets” for each valid query/code pair and thus rank_i is the position of the correct snippet among the distractors. Although no literature is provided, we provide citation for MRR's use at the National Institute of Standards and Technology (NIST) Text REtrieval Conference (TREC) (Radev et al., 2002) signifying its ubiquity in the domain of information retrieval.

NDCG can be defined in parts and all parts assume that a grading scheme has been created to score the relevance of all results. Cumulative Gain (CG) is the sum of the relevance of all results using the established grading scheme. Discounted Cumulative Gain (DCG) augments CG by penalizing relevant results for being ranked lower by logarithmically reducing the grade in proportion to the position of the result. Finally, Normalized Discounted Cumulative Gain (NDCG) further adjusts the metric by normalizing the CG of each position across all queries. For the CodeSearchNet Challenge, the GitHub team had a group of expert programmers rate possible results for 99 pre-defined natural language queries. This process is further detailed in Husain et al. (2019).

MRR over the test split is outputted by an included script once the model has finished training. Unfortunately, the GitHub team prefers that researchers submit their predictions to the W&B leaderboard in order to obtain NDCG results and submissions are capped at one per two weeks. Although this restriction does prevent spam, it made evaluation using NDCG impossible for our project given the time frame. Thus, we chose to only submit our best model to the leaderboard.

Outlined as a key factor of our approach, each of our models is intended to be compared to its most similar baseline counterpart and we do as such in our error analysis. For each training run, the W&B codebase integration generates a table of the MRR results on the Test Set in addition to the actual scores. Namely, for each test example, we have the rank that our models gave the correct code snippet among the 999 distractor snippets. In addition, the examples displayed in the tables are consistent between every run. This, combined with the fact that the GitHub team's runs are also public, means that we can compare the delta of the rankings between models. Using this delta, we present the overall performance difference as well as any specific variation that we noted.

7.1 CBoW NDCG Results and Implications

We would first like to quickly acknowledge some interesting findings we encountered with the CodeSearchNet Challenge leaderboard. As we briefly mentioned in our reporting of the results of the CBoW model, the NDCG scores were not what we expected. The CBoW model scored much better on MRR, but worse on NDCG. It should be noted that there are multiple NBoW submissions with equivalent hyperparameter values and their scores vary slightly. Thus, a possible explanation is that perhaps if we trained the model a few more times, we could obtain a more accurate picture of its performance.

However, we see that this phenomenon also occurs with the SelfAtt model. In their paper, the GitHub team attributes this to the fact that although the SelfAtt model has a high capacity, the NBoW model seems to be effective at keyword matching, clearly an important part of search. Now, with CBoW, we also have a relatively low capacity model that scores well on MRR, but not on NDCG.

There are a myriad of possible interpretations of these results so we will summarize our best ones. Most simply, the one additional layer in CBoW when compared to NBoW could cause the model to loose its ability to learn keyword matching. We also suspect that MRR might be a biased estimator of NDCG in that, for smaller accuracy values, the metric were congruent but the consistency fades as MRR increase. Finally, given that the test set for MRR and NDCG are fixed, certain model types could be overfitting to the specific examples chosen by the GitHub team.

7.2 NBoW with Concat Pooling vs. GitHub's NBoW Baseline

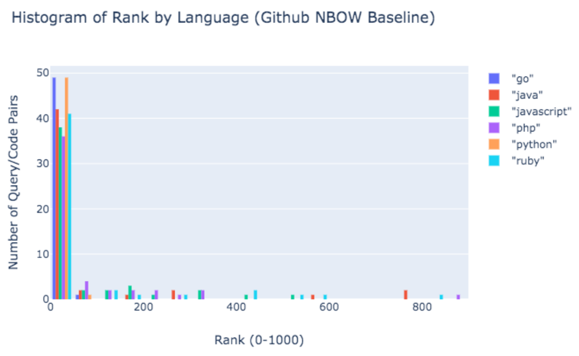

Figure 1: NBoW Baseline Rank Histogram

GitHub's NBOW baseline provides reasonable results. Figure 1 shows the model is able to generalize well, and produce consistent results. Quantitatively, we observe that the model's predictions of rank have a standard deviation of *131.029*. This high values is due to the outliers with extremely high rank.

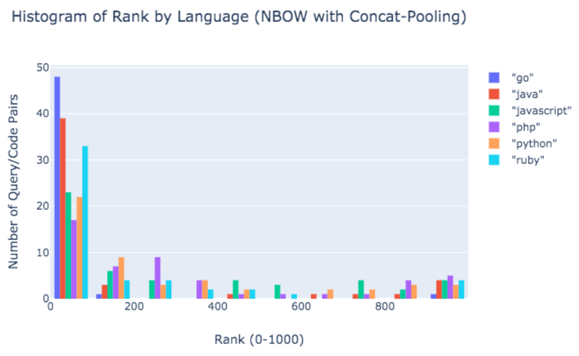

Figure 2: NBoW Concat Pooling Rank Histogram

Our attempt at providing the NBoW model with more capacity led us to the Concat Pooling method. Figure (fig) shows that our initial hypothesis was wrong, and that we actually hindered the model's ability to rank code query pairs. Quantitatively, we observe that with additional Concat Pooling the model's predictions of rank have a standard deviation of *303.5964*.

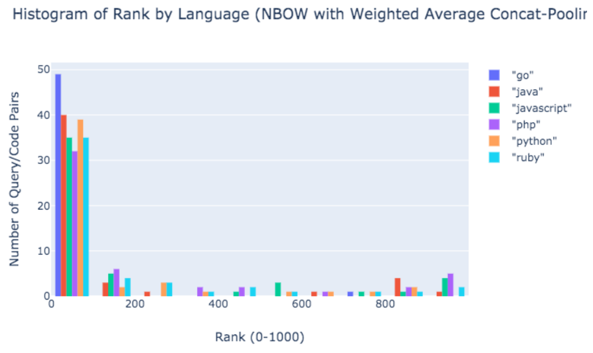

Figure 3: NBoW Weighted Average Concat Pooling Rank Histogram

Using a learned weighted average without Concat Pooling implementation allowed the model to generalize better than the original Concat Pooling method, but still performed poorly in comparison to the original NBoW baseline model. We see quantitatively, this method obtained a standard deviation of *265.9918*, in terms of rank. Specifically, we see that the distribution of outliers is generally suppressed, and that the high-rank examples are more consistent.

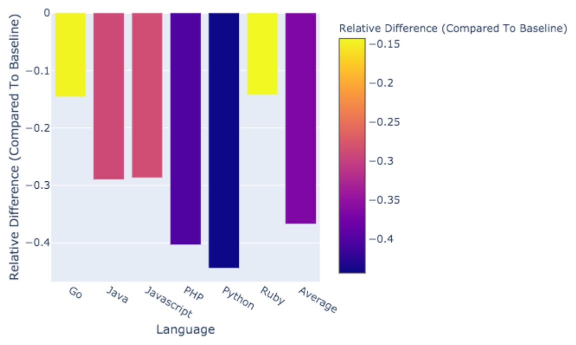

Figure 4: NBoW with Concat Pooling MRR Relative Difference

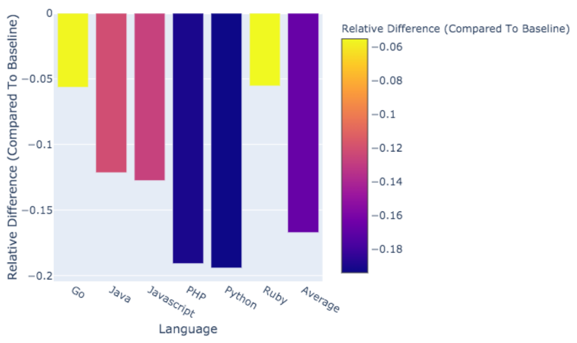

Figure 5: NBoW with Weighted Average Concat Pooling MRR Relative Difference

Figures 4 and 5 show the general down trend of the Concat Pooling used with our NBoW models. Figure 5 shows that even with the additional modification made, to provide the pooling layer with a learned weighted average, did not provide the model with extra contextual capacity as we had hoped.

7.3 biRNN with Concat Pooling vs. GitHub's biRNN Baseline

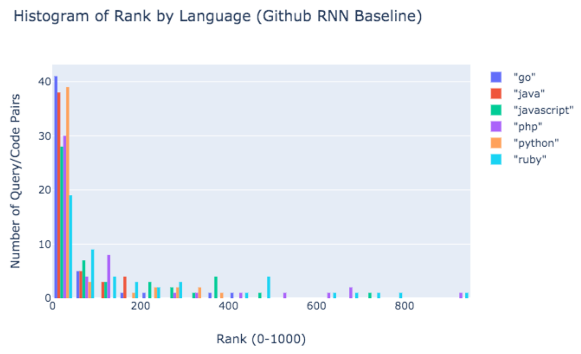

Figure 6: biRNN Baseline Rank Histogram

Figure 6, like the previous analysis, shows the biRNN baseline's rank distribution. The standard deviation of this distribution was *163.5473*. We find it interesting that this value is close to the standard deviation of the NBoW baseline. This would suggest that the models perform similarly poorly on outliers.

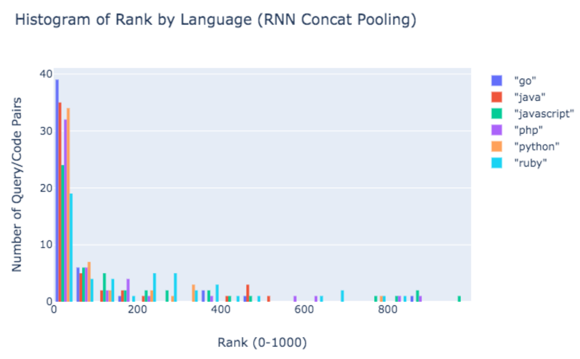

Figure 7: biRNN with Concat Pooling Rank Histogram

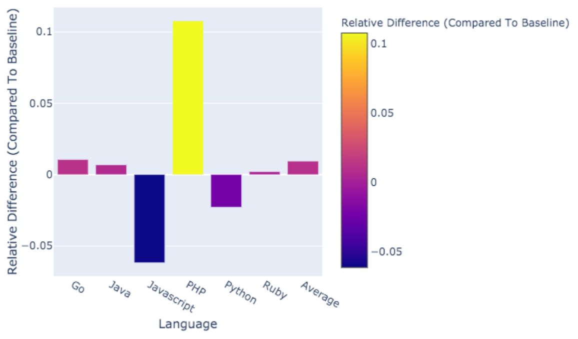

Figure 8: biRNN with Concat Pooling MRR Relative Difference

Although our model scored slightly better on MRR, the standard deviation of the rank distribution of our model was *197.0349*, which is 33.4876 higher than the biRNN baseline at *163.5473*. This would suggest that the additional information provided by the mean and max pool only contributes to better performance on examples that were already given a high rank by the baseline while outliers remained poorly ranked. We posit that if we were to train the model more, we might see a regression to the mean model performance, at or near the performance of the biRNN baseline.

7.4 Our CBoW vs. GitHub's NBoW Baseline

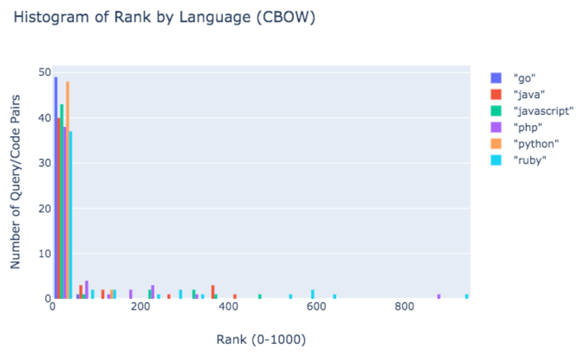

Figure 9: CBoW Rank Histogram

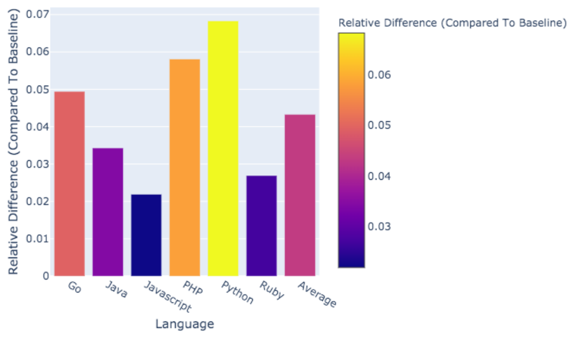

Figure 10: CBow MRR Relative Difference

Our CBoW model beat Github's NBoW baseline MRR in every language and in average. Our smallest relative difference to the NBoW baseline is 0.02, showing an overall significant improvement in terms of MRR. We also see significantly higher individual language improvements in PHP, Python, and Go, all of which have an MRR difference of 0.05 or higher. With a standard deviation of *124.3785* compared to the NBoW baseline standard deviation of *131.029*, the CBoW model not only shows higher performance but also more consistency than NBoW.

8 Contributions of group members

List what each member of the group contributed to this project here.

- Shishir Jakati: attempted ELMo implementation, implemented Weighted Average Concat Pooling, error analysis visualizations, gathered information from W&B, lots of writing

- Sean Kelley: implemented Concat Pooling, helped setup AWS training environment, helped gather rank information from W&B, lots of writing
- Mukul Kudlugi: implemented CBoW, helped determine that Word2Vec approach should be changed, lots of writing
- Sidharth Subramanian: attempted Attention implementation, helped gather W&B data, lots of writing

9 Conclusion

Although it appeared to be straightforward from the outside, our project proved quite challenging. The most formidable ordeal being the wonderful monstrosity that is Tensorflow. It was often exceedingly difficult to find relevant information that we needed to resolve errors with our models and debugging static computational graphs is frustrating at best. All this considered, it was exciting to participate in live research and compare our models to others also working in the field.

Going forward, we hope to have our CBoW leaderboard results merged into the CodeSearchNet repository so our position can be made official. Our results and CBoW model should be of particular interest for others participating in the challenge given the large discrepancy between its MRR and NDCG scores. Related to that, we believe that future work must be done on analyzing the train and test data to ensure that they are roughly representative and thus the metrics are not biased to the data they are given. We also posit that a stochastic method of determining the train and test data as opposed to the fixed sets provided could yield more accurate results. Overall, the subfield of Semantic Code Search still remains in its infancy, but the results of current approaches have already produced interesting debates and now offer countless possible research directions.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Allamanis, M., Peng, H., and Sutton, C. (2016). A convolutional attention network for extreme summarization of source code.

Cambronero, J., Li, H., Kim, S., Sen, K., and Chandra, S. (2019). When Deep Learning Met Code Search. *arXiv e-prints*, page arXiv:1905.03813.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural Language Processing (almost) from Scratch. *arXiv e-prints*, page arXiv:1103.0398.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv e-prints*, page arXiv:1810.04805.

Howard, J. and Ruder, S. (2018). Universal Language Model Fine-tuning for Text Classification. *arXiv e-prints*, page arXiv:1801.06146.

Husain, H. and Wu, H.-H. (2018). Towards natural language semantic code search. `https://github.blog/2018-09-18-towards-natural-language-semantic-code-s` Accessed: 2019-10-1.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv e-prints*, page arXiv:1909.09436.

Merity, S., Shirish Keskar, N., and Socher, R. (2017). Regularizing and Optimizing LSTM Language Models. *arXiv e-prints*, page arXiv:1708.02182.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *arXiv e-prints*, page arXiv:1301.3781.

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Peters, M. E., Ammar, W., Bhagavatula, C., and Power, R. (2017). Semi-supervised sequence tagging with bidirectional language models. *arXiv e-prints*, page arXiv:1705.00108.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. *arXiv e-prints*, page arXiv:1802.05365.

Radev, D. R., Qi, H., Wu, H., and Fan, W. (2002). Evaluating web-based question answering systems. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC'02)*, Las Palmas, Canary Islands - Spain. European Language Resources Association (ELRA).

Sachdev, S., Li, H., Luan, S., Kim, S., Sen, K., and Chandra, S. (2018). Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 31–41. ACM.

	Encoder			CodeSearchNet Corpus (MRR)						
	Text	Code	Pool	Go	Java	JS	PHP	Python	Ruby	Avg
Github	biRNN	biRNN	LWA	0.4524	0.2865	**0.1530**	0.2512	**0.3213**	0.0835	0.4262
Ours	biRNN	biRNN	CP	**0.4630**	**0.2933**	0.0913	**0.3589**	0.2985	**0.0856**	**0.4357**

Table 4: Sequence embedding performance comparison for biRNN model using Mean Reciprocal Rank (MRR) on Test Set of CodeSearchNet Corpus.

	Encoder			CodeSearchNet Corpus (MRR)						
	Text	Code	Pool	Go	Java	JS	PHP	Python	Ruby	Avg
Github	NBoW	NBoW	LWA	**0.6409**	**0.5140**	**0.4607**	**0.4835**	**0.5809**	**0.4285**	**0.6167**
Ours	NBoW	NBoW	CP	0.4949	0.2242	0.174	0.0801	0.1367	0.2858	0.2497
	NBoW	NBoW	CP+WA	0.5846	0.3926	0.3333	0.2928	0.3867	0.3732	0.4495

Table 5: Sequence embedding performance comparison for NBoW model using Mean Reciprocal Rank (MRR) on Test Set of CodeSearchNet Corpus.

	Encoder			CodeSearchNet Corpus (MRR)						
	Text	Code	Pool	Go	Java	JS	PHP	Python	Ruby	Avg
Github	NBoW	NBoW	LWA	0.6409	0.5140	0.4607	0.4835	0.5809	0.4285	0.6167
Ours	CBoW	CBoW	LWA	**0.6903**	**0.5483**	**0.4826**	**0.5416**	**0.6492**	**0.4554**	**0.6600**

Table 6: Token embedding performance comparison for NBoW baseline model versus our CBoW model using Mean Reciprocal Rank (MRR) on Test Set of CodeSearchNet Corpus.

Encoder	Hyperparameter	Value
All	token_vocab_size	10000
	token_vocab_size	10000
	token_vocab_count_threshold	10
	token_embedding_size	128
	seq_embedding_size	128
	use_subtokens	False
	mark_subtoken_end	False
	max_num_tokens	200
	use_bpe	True
	pct_bpe	0.5
NBoW	nbow_pool_mode	'weighted_mean'
RNN	rnn_num_layers	2
	rnn_hidden_dim	64
	rnn_cell_type	'LSTM'
	rnn_is_bidirectional	True
	rnn_dropout_keep_rate	0.8
	rnn_recurrent_dropout_keep_rate	1.0
	rnn_pool_mode	'weighted_mean'
1D-CNN	1dcnn_position_encoding	'learned'
	1dcnn_layer_list	[128, 128, 128]
	1dcnn_kernel_width	[16, 16, 16]
	1dcnn_add_residual_connections	True
	1dcnn_activation	'tanh'
	1dcnn_pool_mode	'weighted_mean'
BERT	self_attention_activation	'gelu'
	self_attention_hidden_size	128
	self_attention_intermediate_size	512
	self_attention_num_layers	3
	self_attention_num_heads	8
	self_attention_pool_mode	'weighted_mean'
D-CNN+BERT	*same as 1D-CNN and BERT*	

Table 7: Baseline Encoder Hyperparameters