Project Report

Dhaval Patel - DJP526
Disha Papneja - DP3074
Akshat Khare - AK7674

Introduction:

Semantic code search is the task of retrieving *relevant code* given a *natural language* query. While related to other information retrieval tasks, it requires bridging the gap between the language used in code (often abbreviated and highly technical) and natural language more suitable to describe vague concepts and ideas.

Here, the neural bag of words model performs very well, whereas the stronger neural models on the training task do less well. We note that the bag of words model is particularly good at keyword matching, which seems to be a crucial facility in implementing search methods. This hypothesis is further validated by the fact that the non-neural ElasticSearch-based baseline performs the best among all baselines models we have tested.

In this notebook we have implemented Neural Bag of Words approach which is a baseline model which can be referenced from:
https://github.com/github/CodeSearchNet

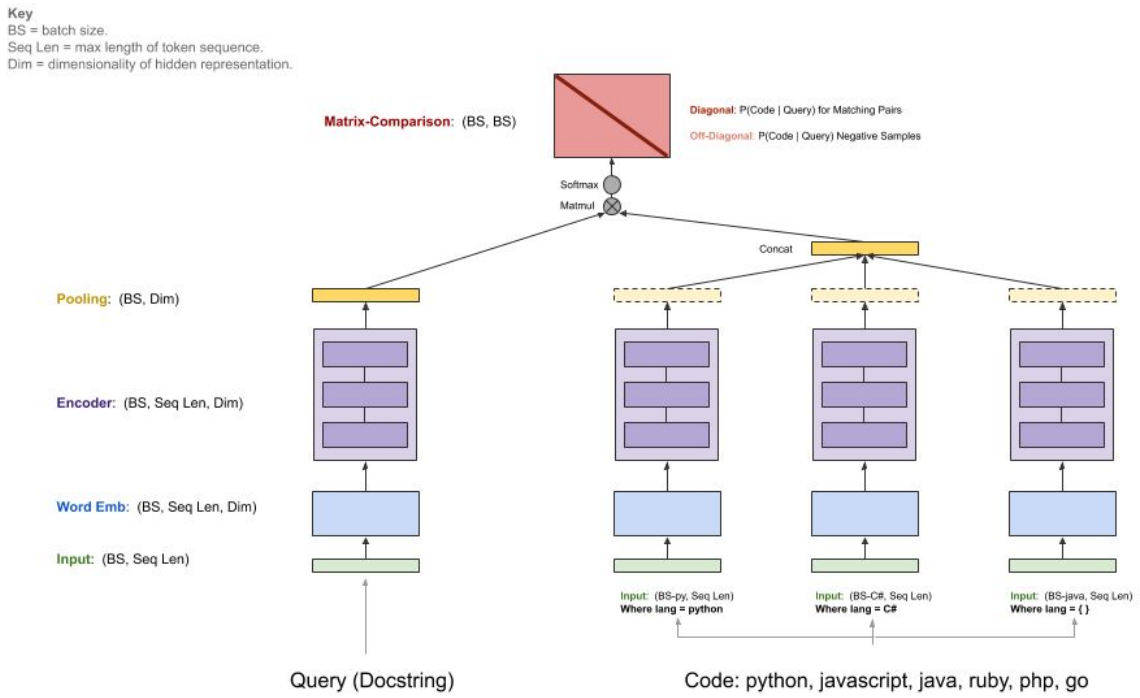
Although the requirement was only for Python, we have done this for all the languages. We needed extra computing power so we requested extra GPUs from Amazon Web Services (AWS).

Data

The primary dataset consists of 2 million (comment, code) pairs from open source libraries. Concretely, a comment is a top-level function or method comment (e.g. docstrings in Python), and code is an entire function or method. Currently, the dataset contains Python, Javascript, Ruby, Go, Java, and PHP code. More information about the dataset can be found here -
https://github.blog/2019-09-26-introducing-the-codesearchnet-challenge/

Baseline Model:

Our baseline models ingest a parallel corpus of (comments, code) and learn to retrieve a code snippet given a natural language query. Specifically, comments are top-level function and method comments (e.g. docstrings in Python), and codeis an entire function or method. Throughout this repo, we refer to the terms docstring and query interchangeably.
The query has a single encoder, whereas each programming language has its own encoder. The available encoders are Neural-Bag-Of-Words, RNN, 1D-CNN, Self-Attention (BERT), and a 1D-CNN+Self-Attention Hybrid.
The diagram below illustrates the general architecture of our baseline models:

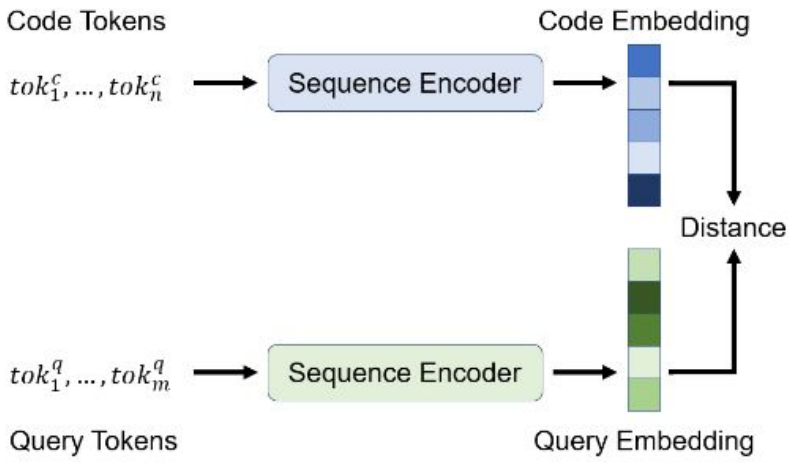


We use joint embeddings of code and queries to implement a neural search system. Our architecture employs one encoder per input (natural or programming) language and trains them to map inputs into a single, joint vector space. Our Training objective is to map code and the corresponding language into vectors that are near to each other, as we can then implementa search method by embedding the query and then returning the set of code snippets that are “near” in embedding space. Although more complex models considering more interactions between queries and code can perform better, generating a single vector perquery/snippet allows for efficient indexing and search.To learn these embedding functions, we combine standard se-quence encoder models in the architecture shown .First, we preprocess the input sequences according to their seman-tics: identifiers appearing in code tokens are split into sub tokens(i.e.a variablecamelCaseyields two sub tokens cameland case),and natural language tokens

are split using byte-pair encoding(BPE)Then, the token sequences are processed to obtain (contextual-ized) token embeddings, using one of the following architectures.

- Neural Bag of Wordswhere each (sub)token is embedded to alearnable embedding (vector representation).
- Bidirectional RNN models where we employ the GRU cell to summarize the input sequence.
- 1D Convolutional Neural Networkover the input sequence of tokens.
- Self-Attention Where multi-head attention is used to com-pute representations of each token in the sequence

The token embeddings are then combined into a sequence embed-ding using a pooling function, for which we have implemented mean/max-pooling and an attention-like weighted sum mechanism.



Here, the neural bag of words model performs very well, whereas the stronger neural models on the training task do less well. We note that the bag of words model is particularly good at keyword matching, which seems to be a crucial facility in implementing search methods. This hypothesis is further validated by the fact that the non-neural ElasticSearch-based baseline performs the best among all baselines models we have tested.

Setup:

We followed the steps below to set up GPU and Docker using AWS:

1. Open an AWS account by using an email ID.
2. Enter your card details.
3. In the AWS Management console search EC2 then go to EC2 dashboard.

4. Select the instance tab, then select launch instance.
5. Select m60 GPU, CPU count to be 16 and Memory 122 EBS
6. We then set up docker.

After setting up docker we ran the following commands to run the baseline model:

```
# clone this repository
git clone https://github.com/github/CodeSearchNet.git
cd CodeSearchNet/
# download data (~3.5GB) from S3; build and run the Docker container
script/setup
# this will drop you into the shell inside a Docker container
script/console
# optional: log in to W&B to see your training metrics,
# track your experiments, and submit your models to the benchmark
wandb login
# verify your setup by training a tiny model
python train.py --testrun
# see other command line options, try a full training run with default values,
# and explore other model variants by extending this baseline script
python train.py --help
python train.py

# generate predictions for model evaluation
python predict.py -r github/CodeSearchNet/0123456 # this is the
org/project_name/run_id
```

Training:

We have a machine, we have a virtual machine, and we are ready to train our model.

First, we do a simple training run with a small amount of data. This lets us know our setup is working. While it is running you will see the in process training run on your wandb dashboard at https://www.wandb.com. After the training run is finished you will see the results of your training run in the wandb dashboard.

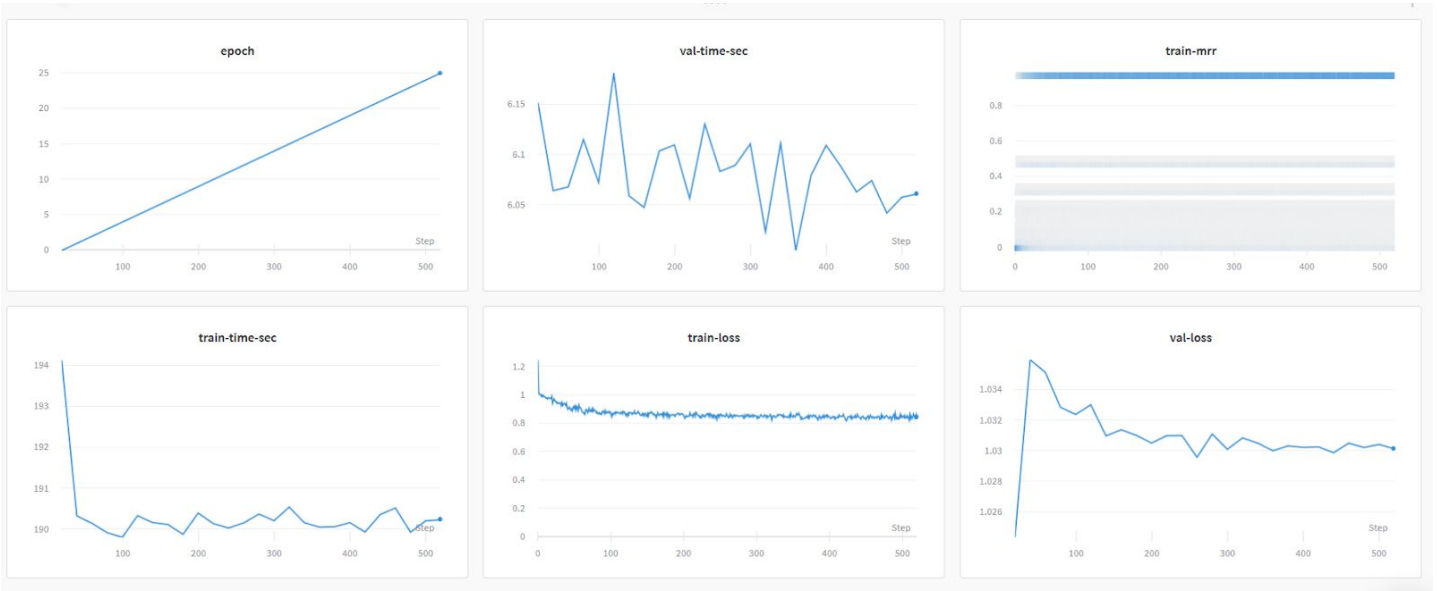
From the Docker shell prompt use python to run train.py with a --testrun flag. (For instructions on how to get to the Docker shell prompt see Part 3 above.)

Input

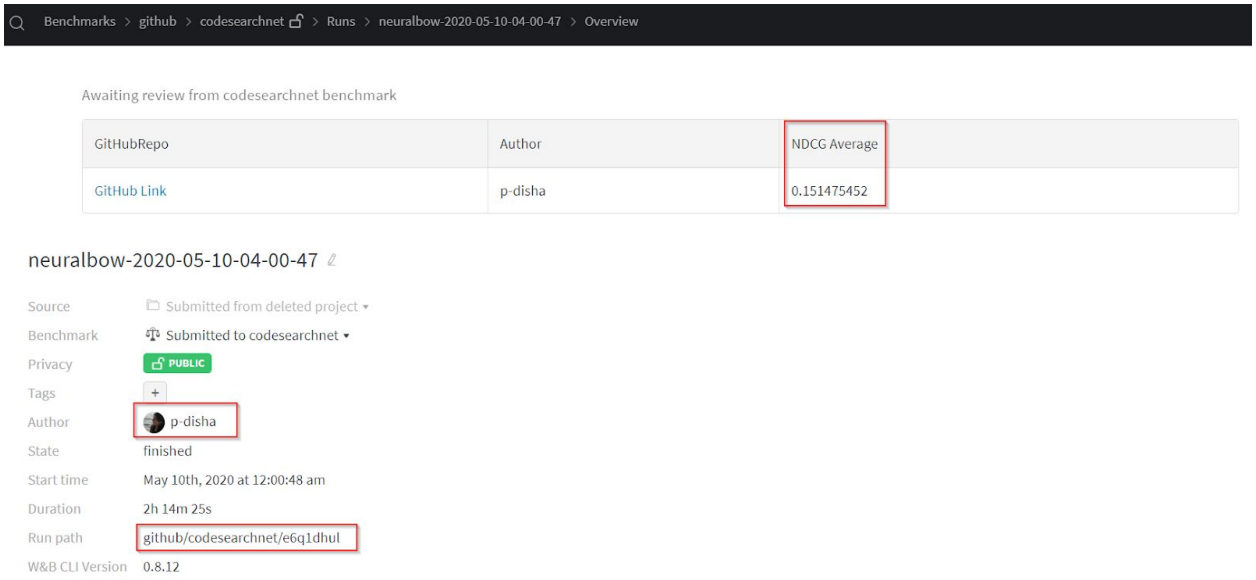
```
# python train.py --testrun
```

This will take about 15 minutes. Once the test run is complete we can go to the weights & Biases website to see a lot of great data about how we did. There is a dashboard that will show you loss, accuracy, and a variety of machine stats.

Cut and paste the URL in the last line of your output into the browser. This will take you to an amazing dashboard with more data about your test run than you had in your wildest data science dreams.



Benchmark Results:



Conclusion:

We can get better accuracy by tuning parameters which are used for training.

Parameters	Description
--max-num-epochs EPOCHS	The maximum number of epochs to run [Default:300]
--model MODELNAME	Choose model type [default: neuralbowmodel]
--test-batch-size SIZE	The size of the batches in which to compute MRR. [default: 1000]

We have also implemented our own bag of words model which can be found here:
https://github.com/dhavalpatel290/AI-Project-3-CS-GY-6613/blob/master/TF_IDF.ipynb

Concluding Remark:

Use of different NLP methods of Document Ranking like RNN, Probabilistics models etc might lead to better accuracy but due to limited implementation and computation time constraints, we were unable to try out other methods.