Université de Montréal

Word Sense Disambiguation

Course: IFT3335 - H21

Professor: Jian-Yun Nie

Rubén Buelvas

May 2021

Content

Introduction

The Word Sense Disambiguation problem is an NLP problem that consists of finding the correct definition of a word given a context. For this project, the target word was “interest”. 6 definitions were given in a text with and without POS tags. Different datasets were created, tested, and used with AI models that were also tested with different configurations. The work consisted mainly of two phases: feature extraction and model training, which will be discussed in this document. The code can be found at GitHub.

Feature extraction

To extract features two files were used, one with POS tags and the other with the text without any processing (only the separator between sentences). In total, 4 datasets were created. For each dataset, different configurations of preprocessing, stopwords, and vectorization were used. The datasets were tested with a Naive Bayes model, as it didn't require any type of configuration.

Natural Words dataset (NW)

The first dataset one was the dataset of the words around the target word. For this dataset stemmezation was used in order to get only the radical of the words and simplify the vocabulary.

Here is the accuracy changing the number of words (context window) and using different configurations.

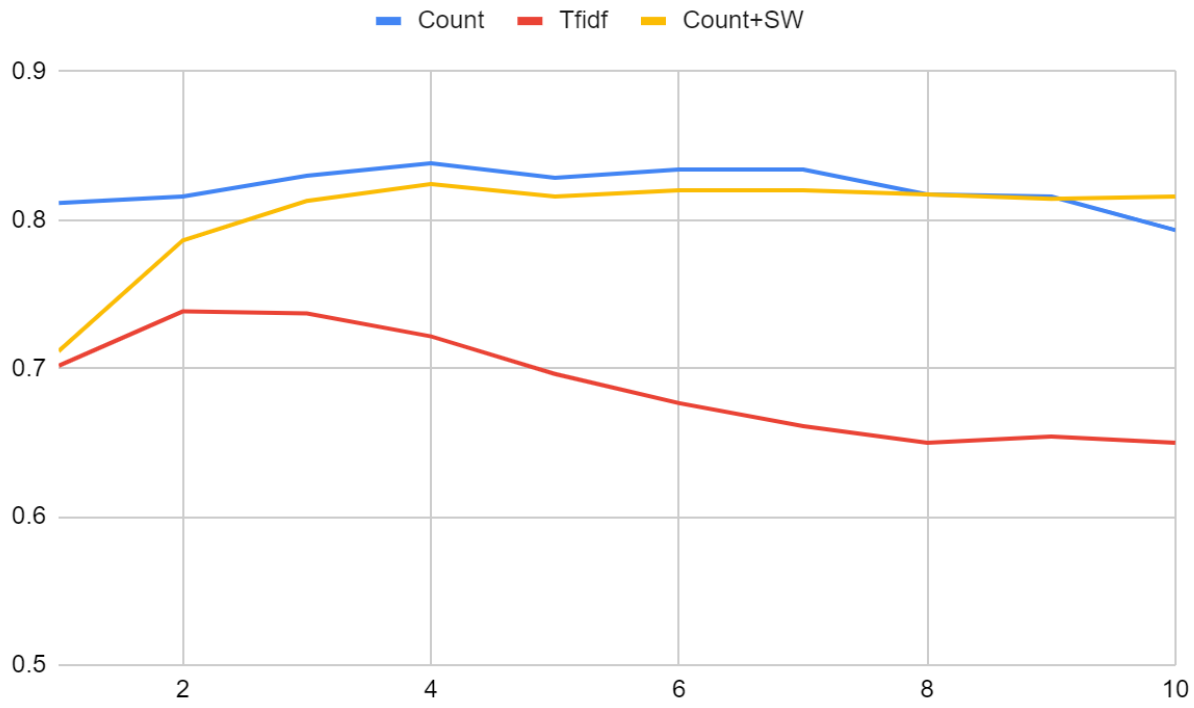

Fig 1: Accuracy against number of words

As expected, the usage of stopwords didn't affect the accuracy of the model using tf-idf. The count vectorizer had better results.

Grammatical Categories dataset (GC)

The second dataset used was made with only the POS tags of the grammatical categories of words. For this one, a custom collection of stopwords was used.

The results weren't good, but the usage of bigrams proved useful to ameliorate the performance of tf-idf.

Here is the accuracy of Naive Bayes changing the number of words and using different configurations:

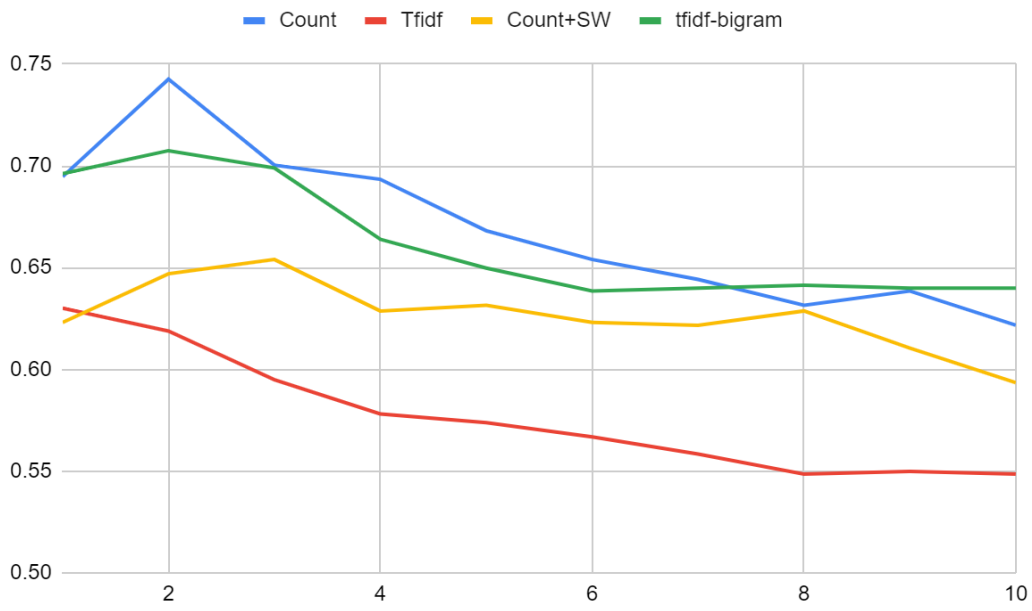

Fig 2: Accuracy against number of words

Whole Sentence dataset (WS)

The third dataset is the collection of natural words, using the whole sentence and adding the target word into the sentence removing the class tag from it. The best configuration proved to be using the count vectorizer with the English set of stopwords from sklearn.

With the best configuration, this dataset had a Naive Bayes accuracy of 0.8256, slightly better than the best of the other datasets until now (0.8242). Here's the accuracy with Naive Bayes compared with the best-case scenario for the other datasets:

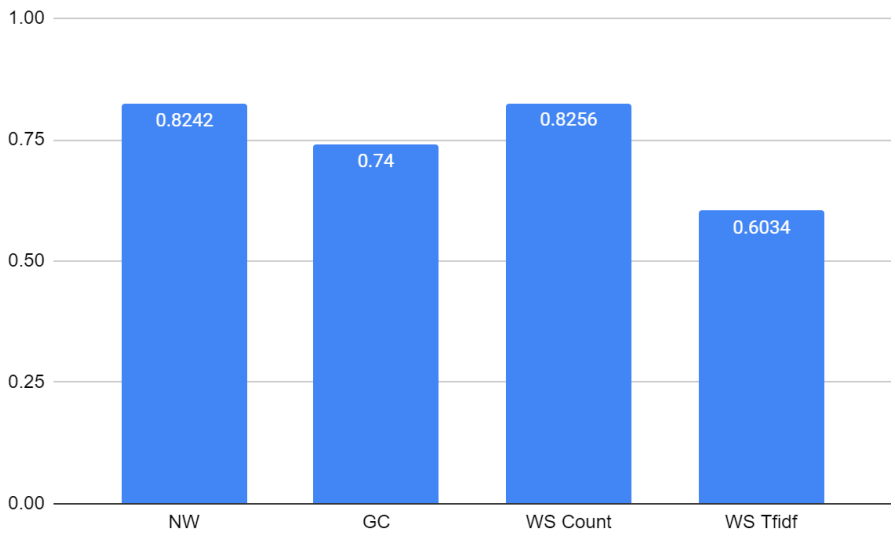

Fig 3: Performance of the Whole Sentence dataset against the rest

WS with GC dataset (WS_w_GC)

The last dataset created was a combination of the WS Dataset and the GC Dataset, saving the word and its POS tag in a bigram.

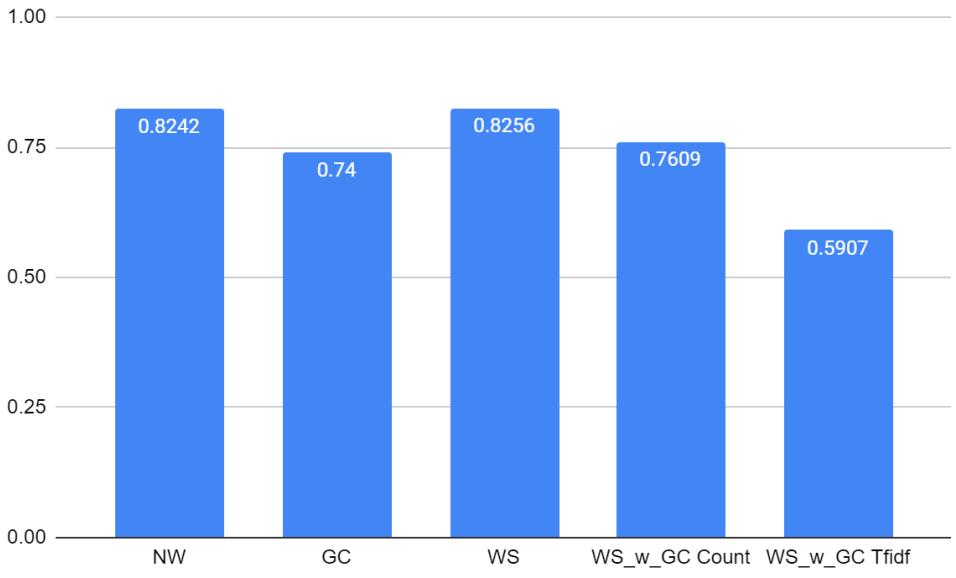

Fig 4: Performance of the WS_w_GC dataset against the rest

Model performance

After getting the best configuration for each dataset, at least for Naive Bayes, they were used to train and test different models.

Naive Bayes

The Naive Bayes model was the first one to be tested, as it didn't have any changeable configuration. and had the following results:

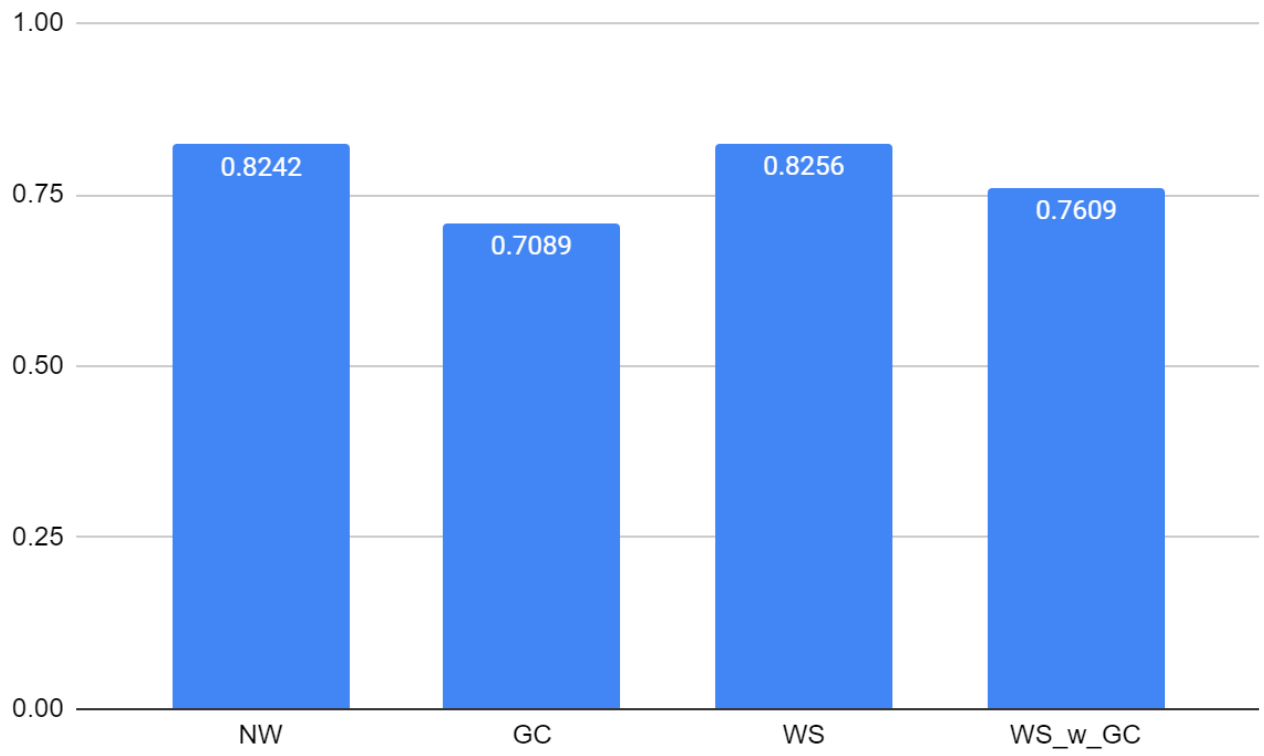

Fig 5: Performance of the Naive Bayes model with all datasets. Accuracy against depth

Decision Tree

The Decision Tree model had a performance very similar to Naive Bayes. It was tested with different depths. Here are the results:

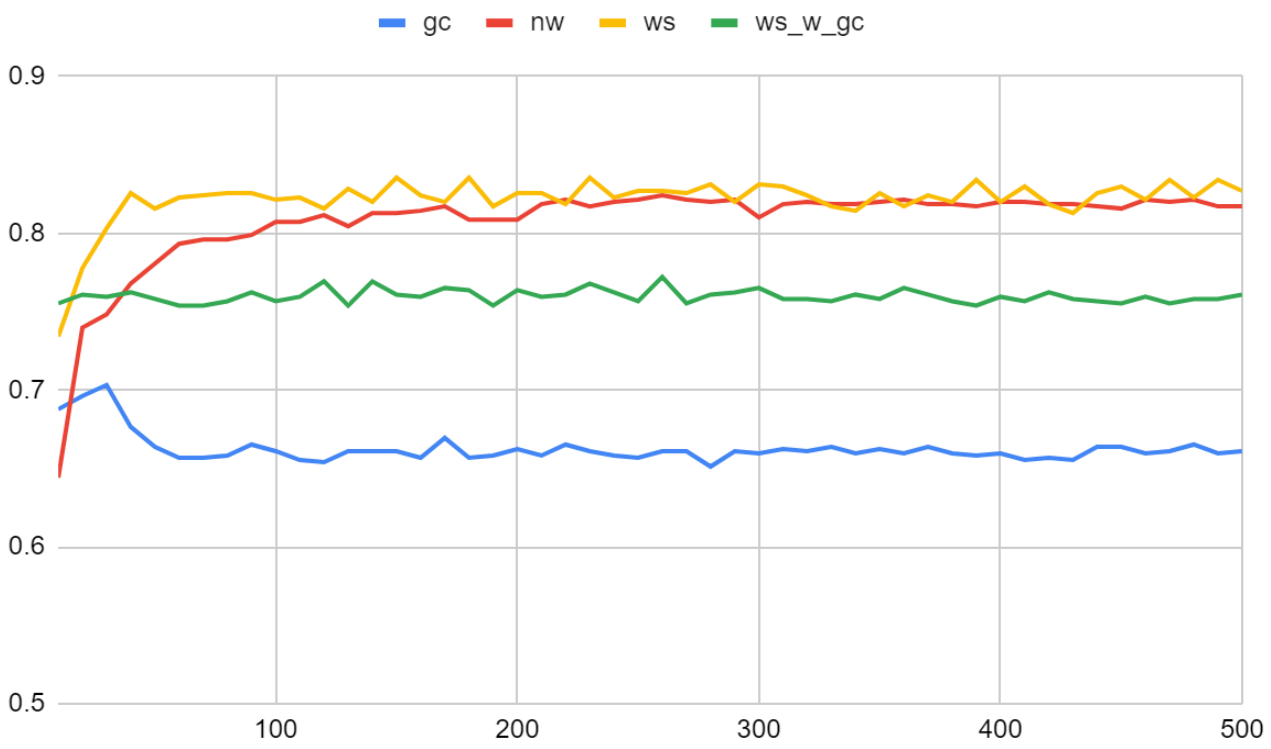

Fig 6: Performance of the Decision Tree model with all datasets. Accuracy against depth

Multi-Layer Perceptron

The multi-layer perceptron was tested with the optimal settings for each dataset changing the size of the hidden layer. For these tests, the optimizer used was adam, because lbfgs had problems converging.

The size of the hidden layer didn't make a significant impact on the performance of the net. The results are the following:

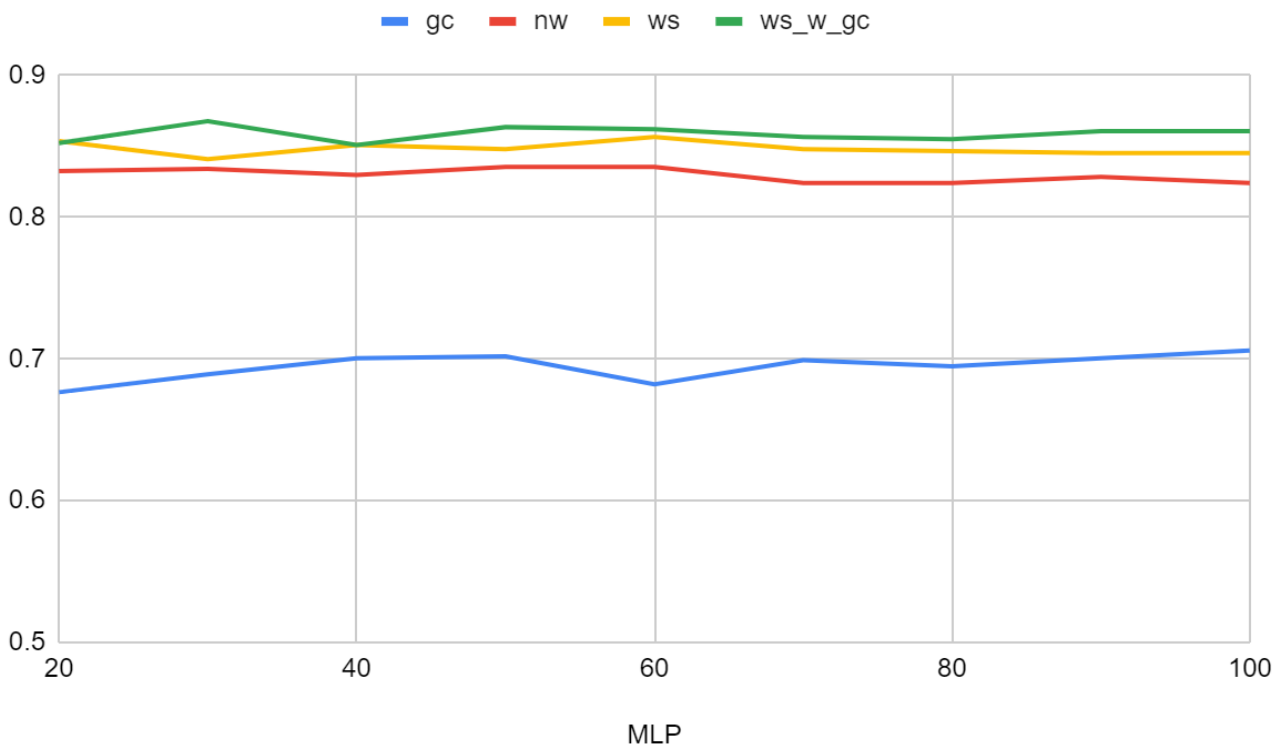

Fig 7: Performance of the MLP with all datasets. Accuracy against size of hidden layer.

Final thoughts

The best configuration was the multi-layer perceptron with the WS_w_GC dataset, achieving an accuracy of 0.8734 in this test. Here are the final results for each model using optimal settings for the models and the datasets:

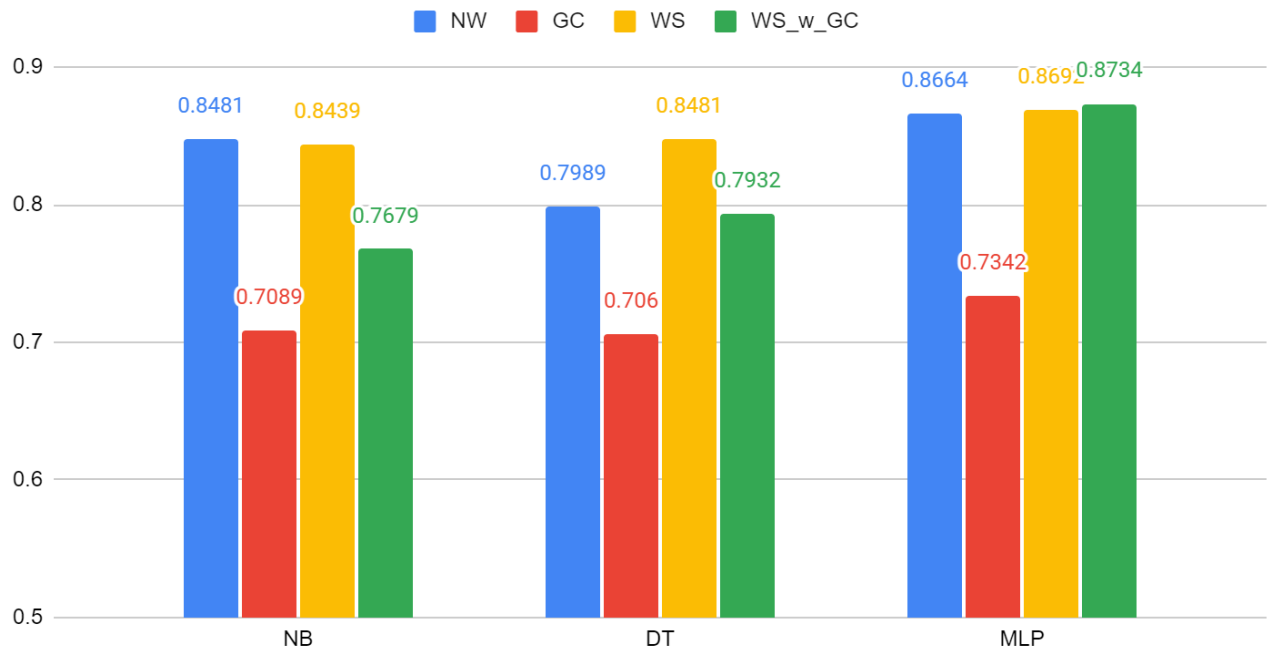

Fig 8: Performance of all models with each dataset.

First of all, is important to note that, even though as expected the best model was the multi-layer perceptron, the other models weren't that bad. Especially considering that in a 6-class classification problem, a random agent is expected to have an accuracy of 0.16.

In addition, even though the initial performance of the WS_w_GC dataset was disappointing, the multi-layer perceptron model was capable of extracting information that the other models couldn't. This could mean that the best way to find the optimal configuration for each dataset wasn't testing with a particular model, as was done in this project.

Finally, It was much more impactful (and at first underrated) the stage of data preprocessing and feature extraction than the configuration of the models themselves. The POS tags were useful but had to be handled carefully, as results of different configurations were often contra-intuitional.

How to run the program

All functionality is contained in the main.py file. The list of requirements can be found in the file requeriments.txt.

We can use the program to generate datasets from the original source (both interest-original.txt and interest.acl94.txt files have to be present in the same folder). The program does the feature extraction and generates 4 .csv files in the same folder with the preprocessed data.

To generate these files, use the following command:

```
python main.py generate_datasets
```

To test the performance of different configurations for the decision tree and multi-layer perceptron models, the program has two functions. The program tests a pre-configured range of settings for each model and then saves the performance results in .xlsx files.

The decision tree function generates results for all datasets at once. To test the configurations for the decision tree model, use the following command:

```
python main.py test_dt_config
```

Testing can take some time, so the function can generate results for only one dataset at a time. To test the configurations for the multi-layer perceptron model, use the following command:

```
python main.py test_mlp_config [dataset]
```

We can replace [dataset] with the words “gc”, “nw”, “ws”, and “ws_w_gc”.

If, nevertheless, we want to test all configurations for all datasets at once, we simply use the command without the second argument:

```
python main.py test_mlp_config
```

Finally, we can test the models with their optimal configuration for each dataset. These configurations can be changed by editing the dictionary at the top to the main.py file. To run the models use the following commands:

```
python main.py run_model [model] [dataset]
```

```
python main.py run_all_models
```

We can replace [dataset] with the words “nb”, “dt”, and “mlp”. Running this command will also generate weight files for the MLP model.