Arun Agarwal

Professor Yin

CIS 4523: Foundations of Machine Learning

October 27th, 2022

System Description

For the CIS 4523 Midterm, students were asked to train a non-deep-learning model to classify whether two sentence instances express the same meaning or not. We were provided a training dataset with 4077 instances, a development set with 724 instances (50% positive, 50% negative), and a test dataset with 1000 instances (50% positive, 50% negative).

In this document, I describe what I did to achieve my final model predictions, including the features I designed, the data preprocessing and feature preprocessing, the algorithms and libraries used, and the experiences and lessons I learned in this project.

My final code for this project is in the Code Folder with the name "FinalResult.ipynb". There are many other files in this GitHub repository that will be discussed at the end. It should be noted that, after 30 days of working on this project, I ended up creating over 5000 different models, and I tried to keep a record of all of them in this repository. Thus, it is probably to the Professor's benefit to just look at my FinalResult.ipynb file; however, I would appreciate it if he recognized the tremendous effort I put in to achieving the best accuracy score for this assignment.

To begin, I have created over 20 features to represent the sentences in each row for this assignment. The functions used to create every single one of these features is neatly written near the top of FinalResult.ipynb in the "Preprocessing Functions" section. The first feature I created

is called cosine_similarity_score and represents exactly the cosine similarity between the two sentences. This was created using the counter_cosine_similarity function. Next, length_similarity is a feature that represents how similar the length of the two sentences are. Overlap_score is a feature that measures the amount of overlap between the two sentences by taking the sum of the values in common between the two, dividing it by the length of each sentence, and taking the difference of each ratio. There is also a feature called overlap2_score which performs the similar procedure, but it instead divides the overlapping word count by the summation of the length of the two sentences. Thus, this feature could not have a value greater than .5. The next feature is called cosine/length ratio, which, as its name suggests, is simply the cosine score divided by the length similarity. The next feature is called cosine_similarity_score2, which computes the cosine similarity in a slightly different manner based on the sentence preprocessing (discussed later). The jaccard_similarity_score feature exactly represents the jaccard similarity between the two sentences, and the lemma_jaccard_score feature does exactly the same except for using a lemmatized version of the sentences. This feature ends up being the best predictor of the classification for the dataset. Overall_sim_score simply takes the average of many of the syntactic feature scores, including cosine_similarity_score2, jaccard_similarity_score, and lemma_jaccard_score. Overall_similarity_path_semantic measures the semantic similarity between the two sentences by an extensive procedure. This process/function exists in the utils folder in the file named semantic_similarity_measures.py. This file also contains the function that produces the overall_similarity_wup_semantic feature, which is the second semantic feature used for my model. I have a third semantic feature as well that is just the average of these two semantic features, but this does not end up getting used in the final model. The remaining features are all variations of n-grams. That is, a count of the overlap in bigrams, trigrams, and quadgrams between

the two sentences. There is also Jaccard distance and cosine similarity scores produced for bigrams, trigrams, and quadgrams. There were also around 30 other features designed throughout the process, but they were all deleted at some point due to poor prediction capability.

I next discuss the preprocessing done on the dataset and the features. While many different things were previously tried to preprocess the dataset, I will only discuss the preprocessing steps outlined and used in FinalResult.ipynb. To begin, for the train, test, and development datasets, I read in the dataset as a pandas dataframe, noting the tab delimiter. I also renamed the columns and turned the classification column to a numeric value rather than a string. From here, I preprocessed the columns of sentences by first making a new column for each sentence called 'Text_Cleaned1' and 'Text_Cleaned2', respectively. These columns contain the unlemmatized list version of the sentences in which stop words and contractions are removed. Stop words are removed using the nltk library and importing wordnet and stopwords. All preprocessing functions are neatly organized/written in the python file called pre_processor.py. I then created two more columns in the dataset called 'lemmatized_text1' and 'lemmatized_text2' for each sentence, respectively. These columns are simply the lemmatized versions of 'Text_Cleaned1' and 'Text_Cleaned2', which is done through use of WordNetLemmatizer from the nltk library. With these columns produced, I could then create all the features that were mentioned in the previous paragraph. However, it should be noted that, due to the length and complexity of the algorithms for obtaining semantic similarity scores, I needed to split up the dataset and get the scores for each subset to allow my computer to process the data. I then recombined the subsets to get the original dataset. This was only done for the training dataset as it was the only one that was too large for my computer to handle. In doing this, a column called 'scores' was created that immediately got dropped after recombining the subsets. Similarly, when producing the ngram features, columns for

holding the lists the bigrams, trigrams, and quadgrams for each sentence got deleted after the features that used these columns were produced. Another feature preprocessing step completed was to invert the overlap_score and overlap2_score features so that they would match the scale of the other features; that is a higher score meant a better chance that the row instance should be classified as 1. Moving on, another feature preprocessing step I took was to normalize just the semantic features as their values were the only ones that went above the 0 to 1 score range. In addition, I created versions of the training dataset that removed outliers as I felt that this may improve the score. To do this, I removed row instances that did not fall within the .00 to .90 quantile for those classified as 0 and within the .10 to 1 quantile for those classified as 1. This was done based on the column values for only the best predicting feature columns, which was determined after training the first few thousand models. After this point, I dropped all unneeded columns from the dataset, including ‘id’, ‘sentence1’, ‘sentence2’, 'Text_Cleaned1', 'Text_Cleaned2', 'lemmatized_text1', and 'lemmatized_text2'. That is, the final versions of all my datasets only contained numerical scores or a classification value. The only exception to this is the test dataset, which still contains the ‘id’ column as it was required for the final text file output. Other than the outlier removal, all of the above pre-processing steps were completed for the development and test datasets as well. From here, a lot of exploratory data analysis (eda) was done to better understand the scores, and only the most important results are still provided in FinalResult.ipynb. This includes information about the columns (datatype, null count, overall count, mean, standard deviation, minimum, maximum, etc.), histogram plots for all of the features, pairplots for each possible pair of features, a heatmap, and a scatterplot that only shows two features that seemed to best represent the dataset. I later create versions of the datasets that normalize all features; however, these do not get used in the final model.

I next discuss the algorithms and libraries used. All of the libraries used are listed at the top of FinalResult.ipynb, including those that exist in the other code files. For my basic imports, I have pandas for everything dealing with dataframes, numpy for certain numerical operations, nltk for word processing, various imports from nltk (corpis, stem, stem.wordnet), re for removing contractions through regex, math for mathematical operations, and collections for the Counter object (which was necessary for many features). Moving on, for my preprocessing imports, I have StandardScaler (which does not end up being used), sklearn's CountVectorizer for one of my features, sklearn.preprocessing for a certain function I used previously, sklearn.feature_extraction for the text function, pickle, warnings (to be able to ignore warnings), random (to import shuffle and set a random seed), MinMaxScaler for normalization, and various functions from imblearn for attempts at oversampling and undersampling the data. For plotting imports (in the exploratory data analysis step), I only used matplotlib and seaborn. In terms of model imports, many are provided, but the only one that ends up being used in the final result is SVM. However, I would like to note that I tried using Logistic Regression, Decision Tree Classifier, Random Forest Classifier, AdaBoost Classifier, KNeighbors Classifier, and SGD Classifier from sklearn. I also tried using XGB Classifier from xgboost. Some other model imports I had include sklearn's model_section for train_test_split, KFold, and GridSearchCV. Finally, for metric imports, I had all of the common scores, including MSE, MAE, R^2, accuracy, F1, precision, confusion matrix, and ROC_AUC. For my final model, I received the best results with using a tuned SVM model on the dataset that used the top 10 best predicting features ('lemma_jaccard_score', 'overall_sim_score', 'cosine/length_ratio', 'cosine_similarity_trigrams', 'jaccard_similarity_score', 'overlap_score', 'overlap2_score', 'overall_similarity_combined_semantic', 'jaccard_distance_bigrams', 'trigram_similarity'). This model had class weights set to 76 for 0 classification and 24 for 1

classification (since the training dataset was imbalanced as such), a C parameter of 5, a gamma value of 1, and a kernel of rbf. The model is created with a prediction outputted for the validation output in my tuned_svm_classify2 function. I then use the model to make the prediction on the test set, which also only contained the same best 10 features. This prediction is then stored in a text file as outlined in the instructions.

To conclude, I had many great experiences and learning opportunities from this project. Overall, I learned a lot about paraphrase identification as I read over two dozen papers on the subject. I also learned a lot about oversampling and under sampling due to my belief that it would help with this assignment. In addition, I learned a lot about hyperparameter tuning because for all the different models I created, I had to tune the parameters in different ways, which required me to read a lot of documentation. I also fortified my knowledge of almost every non-deep-learning model due to trial and error for this assignment. One of my takeaways from this project is that it is a lot better to create a function that encompasses everything I want/need to test at the beginning rather than go at it step by step, as this led to a lot more testing and forgetting of past results. I also learned the importance of code organization because, after creating the first few thousand models, it became obvious that code organization was necessary to keep track of everything done. Overall, I really enjoyed working on the Midterm project, and I plan to continue working on it after I have submitted due to pure curiosity in trying to improve my accuracy score.