Stephen Hullender
December 6, 2022
CIS 4526, section 1

Final Report: Paraphrase Detection using Deep Learning

Source Libraries and Imports

For the Paraphrase Detection project, all work was done using Google CoLab (Python), with a significant amount of imports sourcing from machine learning and data science libraries. The essentials include pandas, for data analysis and establishing DataFrame objects for data storage and manipulation purposes, NumPy, for supporting mathematical operations and collection of data, and matplotlib, for data visualizations of feature selection and model accuracy or loss. scikit-learn, or sklearn for short, was used to determine how well the metrics were when developing/testing and executing the models.

Scikit-learn was used to calculate each type of metric, including accuracy, precision, recall, and F1 scores. The F1 score was the most important metric as the project observes the results of the F1 scores instead of the accuracy score. From sklearn, other functions include scaling for converting each word into a numerical feature. MinMaxScaler was used to perform scaling based on the minimum and maximum range of features provided from both the vectorized data and additional features. train_test_split is another method that was used to test the data against each other, primarily used for the training and validation data. sklearn also provides a vectorization method using TF-IDF, or Term Frequency Inverse Document Frequency, which takes textual data and converts it to match how frequent the word is present in a given document. Lastly, sklearn is used for Euclidean distance, which is a feature used to determine how far apart two compared words are in relationship to each other.

For natural language processing, a greater area of work was done with the Natural Language Toolkit (NLTK). Natural language processing is important because the types of data we're working with is text-based, and having the words processed through the models requires a significant amount of filtering to ensure the value of each word does not skew results. This means watching out for stopwords, punctuation, contractions, tokenization, and lemmatization. NLTK also provides methods for features like BLEU, NIST, and METEOR. Another library was used to compute one of the features; jiwer was imported to compute word error rate, which is important as it provides features that indicate how many operations it takes to match one word to another via insertions, deletions, and/or substitutions.

Other imports were used primarily for reasons including system performance, regular expression matches, and calculating timestamps for result files. One important library that was added while working on high-performance algorithms was functools; an algorithm for one of the features was optimized using lru_cache, which uses memoization and speeds up the performance of said algorithm.The reason for this import was to receive results quickly, otherwise the algorithm would crash or take at least more than 10 minutes. Finally, importing Google Drive was required since the files would be sourced from a folder inside my personal Google Drive account, and this would also hold the files for test and final results.

Data Preprocessing and Cleaning

For this project, there were 3 files provided for the data: “train_with_label.txt”, the training data, “dev_with_label.txt” for validation, and “test_without_label.txt” for testing data. Each file was a text file, each tuple documenting two sentences and, for training and validation, a label that showed if the two documents were presented as similar or not in paraphrasing. Because a text file format is not optimal in data manipulation, we convert each file to a readable format using pandas. To do this, each file is parsed through, and each line is iterated and distinguished using a tab delimiter. One error that was corrected from the previous project was delimiting lines that contained more than one tab character, a step that, if overlooked, causes sentences to be skipped over or the worksheet to stop due to an Exception/Error.

The first step to cleaning up the text data after distinguishing each document to their respective columns in each DataFrame object, the first step would be to convert each document into an array, as we want to get the statistics of the document based on each word. With that, we converted all words to lowercase to reduce errors. For every word in each document, we replace contractions into their respective words; these can include words like “haven’t”, which turn to “have not”, or “we’ve” into “we have”. This is important since punctuation will affect the value of the words if not inspected; “weve”, from “we’ve” will not make sense and will render the accuracy results lower than potentially expected. Punctuation is filtered through regular expressions on our next step. To double-check that words are purely textual, we also looked out for HTML links/components, spacing, or escape characters such as the tab ‘\t’. Once the words are purified, we take out any words that are redundant or unnecessary in value - stopwords like “and”, “a”, “an” are not needed. We then return a tokenized form of the sentence using WordPunctTokenizer from NLTK. This tokenizer method converts an array of text into numerical values in order to assert that all words are separate.

Lemmatization is another method of natural language processing that involves reading the word and returning the origin word. For example, “happiness” can theoretically return a word like “happy” since that is where the former word takes its meaning. By using lemmatization, we can reduce the variation of words that are similar in meaning, keeping each word valuable.

Once the words are cleaned up, we move on to TF-IDF vectorization. TD-IDF, once again known as Term Frequency - Inverse Document Frequency, calculates the value of each word and its importance by weighing the commonality of a certain entry against all words in the document. The higher the value, the rarer and more valuable the word is, out of a scale of 0 through 1. Vectorization is implemented for all datasets as some form of numerical value needs to replace the text data. One issue that came about when implementing this was the realization that certain words are not shown when comparing some documents. To make sure all words are counted for each pair of documents, we compared the feature names of one document and calculated the set difference against the column for the latter half of the document pair. Any words that were missing from either document were implemented, with a score of 0 for any words that were originally unavailable.

Feature Selection

Several features were implemented in order to accurately predict results based on the training and validation data provided. Most of the features were introduced after the previous midterm project while others were kept due to their reliability in predicting reasonable estimates. Features included the following: overlapping words, Jaccard similarity score, Cosine similarity score, BLEU scores (4 total, based on the distribution and quantity of weights), METEOR score, NIST scores (3 total, similar to BLEU with each score weighted by level of n-gram provided), bigram features, trigram features, Levenshtein distance, Euclidean distance, and word error rate (5 total, with the common word error rate, amount of loss, number of substitutions, deletions, and insertions for each document).

For our first feature, overlapping words calculates how much of each pair of documents overlaps with each other. By taking each pair of documents, we calculate how many words are similar between the two documents using set intersection. Taking the length of each document in the pair, we divide the length of overlapping words by the selected document and return the absolute value of the difference of these divided results. The goal is to figure out how different the overlaps are between two documents. Most of the results return a decimal value closer to 0.0, meaning that most of the document pairs were probably similar. One potential issue that may be presented from these results is that the use of combining missing words from each given document may cause some results to be skewed, but it was decided that this would be ignored anyways.

Next, we calculate the Jaccard similarity score by taking the set intersection and set union of the two document pairs, then returning the percentage of length of intersected words over the words that came from the set union. Jaccard similarity serves to compare the documents in order to figure out which words are shared and/or distinct. Another feature was the cosine similarity score. This feature uses a method from SciPy, a Python library that specializes in computation, named scipy.spatial.distance.cosine, which returns the distance of the two words. A returned score would return 0 for most optimal results, while a 1 means that the two words are completely different in relationship. Since we are looking for similarity scores, we return the opposite by taking the distance score from 1 and returning the inverse, making 1 the closest result and 0 the farthest similarity score. Cosine similarity score is calculated by measuring the cosine of the angle between two words and measuring how far the angle needs to be to point one word in the same direction as the other.

BLEU scores were next on the list of features used, with a method from NLTK used to calculate each score. Since we were working with 4 different types of BLEU, being BLEU-1, BLEU-2, BLEU-3, and BLEU-4, each score was tuned based on the distribution of weights. FOr instance, BLEU-1 would have 1 weight with a value of 1, while BLEU-4 would contain 4 weights with a value of 0.25 each. Since almost all the values had a high number of decimal values, I rounded all values to 12 decimal places, with a reasonable assumption that after that number of decimal digits, the value wouldn't change significantly when measuring BLEU scores later on.

METEOR was the next feature, and comprises calculations involving precision and recall. The purpose of this metric is used to compare how each pair of documents are worded in order of appearance, and how much alignment there is between the given ordering of words. This is crucial if a pair of sentences has the same amount of words but can have different context. NIST scores were used to evaluate the quality of translation between two documents.

Bigram and trigram features were also used to determine how many pairs of words were common between the two documents. Levenshtein distance and Euclidean distance were both used for purposes of, although it was eventually found that Euclidean had little to no influence since all the results were the same (Euclidean was eventually kept in anyways).

Finally, word error rate was another feature that I personally wanted to test to see if any influence would be made. This feature focuses on the error rate, as well as the amount of loss generated when calculating the error rate, the number of insertions, deletions, and substitutions made to generate the same order of words from one document to another.

Algorithms and Models

There are primarily two models that were used in predicting the F1 scores. The simpler model was used with sklearn, specifically the MLPClassifier module. By default, sklearn's implementation of this model uses LBFGS or stochastic gradient descent to optimize performance. The model starts by taking in a randomly-picked sample portion of the training data and comparing it against itself. Then, the model is tested again with the validation data to determine if the accuracy scores measure well comparatively. This is solely for testing, as the goal is to construct a neural network.

The final model was made with Keras. The model is composed of Dense layers that use ReLU and sigmoid functions. The model is also modeled using a Keras component named Sequential, With this model, we compile it with the following parameters: 'sgd', or stochastic gradient descent, for optimizer (in similarity with the MLPClassifier), 'binary_crossentropy' for loss, and 'accuracy' for metrics. All other metrics found via Keras documentation were primarily focused on other metrics like loss function. One other option for testing the accuracy came from a Keras class named BinaryAccuracy, but this was redacted after the results showed mostly accuracy scores of 30% and F1 scores of 20-25%.

While working on the Keras model, some variables such as batch size and number of epochs were tested. With batch size, my model began with 57, following a tutorial on how to set up the Keras MLP model. This was later tested with the following values for batch size: [25, 50]. By having various values of batch size, the purpose is to see how much accuracy each epoch can take; it is estimated that a lower size is better for datasets that are larger. At first, it was planned that more batch size values would be provided, but due to time constraints, it was decided that 25 would be a good value to start. The number of epochs was, also set by default, 1000 at first, but was tested later to include the following values: [100, 500, 1200, 3000]. It is estimated that a larger epoch size will result in more accuracy but a greater loss. It is also important that the worksheet can maintain a stable amount of RAM while operating; at one point, out of curiosity, I decided to run the program with 10,000 epochs to see how much change in accuracy and loss could be visualized, and after around 3,800 epochs, Google CoLab crashed with a stack overflow error.

Results

In general, most of the F1 scores for the MLPClassifier model did not vary a lot, with about an 85%-95% range coming in for both accuracy and F1 scores. For example, the results were given when running MLPClassifier on training data against training data:

Accuracy: 90.5189% **Recall: 90.5189%**
Precision: 91.0298% **F1-score: 90.6891%**

For results on using the testing data through the same model, the following results were provided in the same run:

Accuracy: 89.3250% **Recall: 89.3250%**
Precision: 90.3064% **F1-score: 89.6045%**

When using the Keras model, at first, the results varied a lot each time the model was executed, with some results being similar to MLPClassifier at a higher average of about 90% in F1 scores, and other results showing 75-80% in F1 scores. This was initially caused by the number of epochs and the batch size provided. Therefore, tuning the number of epochs and batch size were crucial in generating better accuracy results. The best accuracy and F1 scores were made from batch size and number of epochs being 50 and 3000, respectively.

Response

Compared to the midterm project, I was much more prepared and knowledgeable in understanding what features to use in the project, and spend time researching other libraries outside of sklearn, like Keras and PyTorch, to familiarize myself with instructions on how to create a full neural network for the Multi-layer Perceptron. Although PyTorch was initially considered, I found Keras to be simpler in comparison while keeping a library that allowed for a more robust implementation of the multilayer perceptron neural network.