Features/Data Processing

All of the features and data preprocessing features are the same as what were used in my midterm project. Given that my previous midterm model was reasonably refined, focusing much attention on the features would be a poor way to spend time in terms of the change in results it would yield. Most of the room for improvement in f1 score comes from improvements to the model processing the features, not the features themselves.

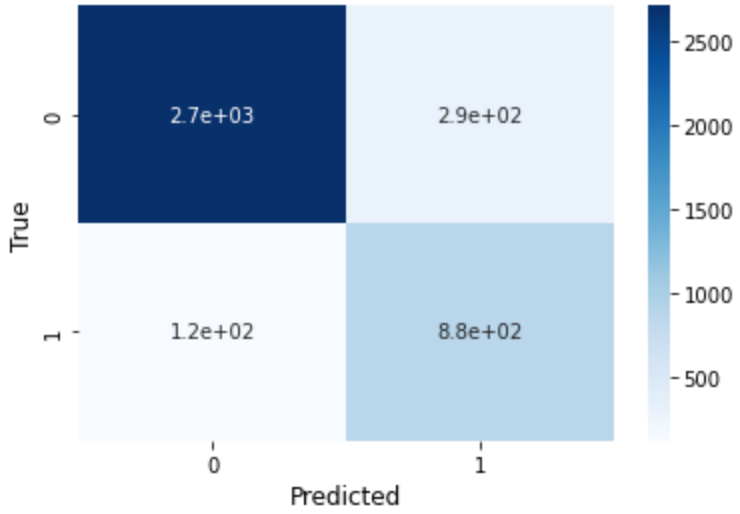
Model Design

This is where the largest piece of the effort went into this project. I tested a few different libraries for their models, but the library that I settled on was keras. Specifically, I used keras models/layers through the tensorflow library. Keras already had a basic setup for how to string layers together, and simple multi-layer perceptrons wouldn't use any other layers than the "Dense" layer defined by keras. Here is what my model looks like:

```
def build_clf(unit1, unit2, learning_rate):
  initializer = GlorotNormal()
  ann = tf.keras.models.Sequential()
  ann.add(InputLayer(input_shape=(13, )))
  ann.add(tf.keras.layers.Dense(units=unit1, activation='relu', kernel_initializer=initializer))
  ann.add(Dropout(0.2))
  if(unit2 != 0):
      ann.add(tf.keras.layers.Dense(units=unit2, activation='relu', kernel_initializer=initializer))
      ann.add(Dropout(0.2))
  ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
  sgd = SGD(learning_rate=learning_rate, momentum=0.9)
  ann.compile(optimizer = sgd, loss = 'binary_crossentropy', metrics = ['accuracy'])
  return ann
```

Initially, I just started with just the final dense layer. This represents a basic logistic regression model, where the features are taken in and converted into a category by the sigmoid function. I did this very simple model initially to test some of the training parameters in the "compile" part of the function, meaning the optimizer, loss_functions, and training metrics. I knew I wanted to use Stochastic Gradient Descent based on the discussions from the slides in class, but I also tested AdaGrad as well, with similar results. The other metrics such as the loss function and metrics seemed to have very little impact as well, and including them would have blown up the number of hyperparameters to tune on to an unmanageable degree for my hardware. After these basic tests, I played around with adding/removing more dense layers with varying width, referred to as "units" keras. I did not see much significant change between adding more layers, until I added in the confusion matrix/heatmaps.

```
Out[142]: Text(33.0, 0.5, 'True')
```



Initially I was just checking for the accuracy score, but when I switched to f1 score and still noticed little change, I realized that the underlying components of the f1 score, namely the precision and recall, could be changing. Something I noticed quickly was that adding more layers made my chance of incorrectly predicted false paraphrase pairs dropped to almost nothing, less than 20 out of the whole 4000 inputs. However, the incorrect number of true paraphrase pairs I guessed was much higher. Using fewer layers added more incorrect predictions, but also more accuracy for the pairs that were paraphrases. I deemed that despite the similar F1 score, this was a better model overall, since it did not overrule the importance of the accuracy of the minority class. This became apparent when I started using a grid search to test for the best model instead of manually altering:

```
params={'mlp__batch_size':[#100, 20, 50, 25,
                           32],
        'mlp__nb_epoch':[200, 100, 300, 400],
        'mlp__unit1':[5,6, 10, 11, 12, 15, 30],
        'mlp__unit2':[5,6, 10, 11, 12, 15, 30, 0],
        'mlp__learning_rate':[0.1,0.01,0.001],

        }

pipe = Pipeline(steps=[('std_scl', StandardScaler()),
                       ('mlp', KerasClassifier(build_fn=build_clf))])

gs = GridSearchCV(pipe, params)
# now fit the dataset to the GridSearchCV object.
gs = gs.fit(X, y)
```

There are a very large number of parameters to tune, which is why I very early on wrote off including more than 2 hidden layers, as the amount of time required to test would be too large for any potential f1 score increases. The batch size parameter was left over from when I was

testing other models than the Stochastic Gradient Descent model, which by default uses a batch_size of 1. The epoch parameter, which represents the number of times the model will work through the training data when optimizing, had a range of values chosen from research of the literature, and while higher values and lower values were tested, the hundreds seemed like the sweet spot. As discussed previously, the unit parameters detail how wide each of the hidden dense layers should be. For the second hidden dense layer, I added an option for a unit of 0, which would skip including the layer. Finally, the learning rate parameter was for the SGD part of the model's training. This was the result of my grid search for the best parameters:

```
{'mlp__batch_size': 32, 'mlp__learning_rate': 0.1, 'mlp__nb_epoch': 200, 'mlp__unit1': 30, 'mlp__unit2': 0}
0.9056539058685302
```

The 0.905 below represents its accuracy on the training set, which does not appear to be over/undertrained. An interesting point is that the best model was one with only one hidden and dropout layer between my manually defined features and the final sigmoid activation layer. These were my scores on the dev set after it was ran through my model:

```
In [140]: precision, recall, fscore, _ = precision_recall_fscore_support(ydev, dev_predicted,
                                                                          average='binary')
          print("Precision:", np.round(precision, 2))
          print("Recall:", np.round(recall, 2))
          # print("F-Score:", np.round(fscore, 2))

          Precision: 0.75
          Recall: 0.88
```

```
In [141]: print("Macro f1 score: ",f1_score(ydev, dev_predicted, average='macro'))
          print("Micro f1 score: ",f1_score(ydev, dev_predicted, average='micro'))
          print("Weighted f1 score: ",f1_score(ydev, dev_predicted, average='weighted'))

          Macro f1 score:  0.8700656231985471
          Micro f1 score:  0.8975
          Weighted f1 score:  0.8999180462686809
```

The file that I submitted for the rehearsal submission had an f1 score of around 0.81, which was lower than my dev scores, but it is somewhat expected from the previous project that there was a dropoff between these two datasets, as our training data is not so large that small quirks in the data that appears in the test would appear in during our model's training.

Algorithms and Libraries

- Csv
  - Used for the constants which affected how files were read from
- Itertools
  - Zip_longest
    - Used in feature creation to iterate through the two sentences to be compared simultaneously
- Warnings
  - Filterwarnings
    - Used to ignore all of the warnings that come from BLEU scores with 0 as output
- Nltk

    - Word_tokenize
        - Used to split the input sentences from files for processing
    - Translate.bleu_score.sentence_bleu
        - Used to calculate BLEU scores for sentence pairs
    - Translate.meteor_score.single_meteor_score
        - Used to calculate meteor score for sentence pairs
    - Ngrams
        - Used to extract ngrams of varying lengths from sentence strings
- Pandas
    - DataFrame
        - Used to store data read from files
    - Read_csv
        - Used to read the data in from the train/dev/test sets
- Sklearn
    - GridSearchCV
        - Used to test different hyperparameters for tuning model
    - Accuracy_score
        - Used to print the accuracy score of the model on the training/test set to see if model is overtrained
    - Precision_recall_fscore_support
        - Used to print f score and f score components
    - Confusion_matrix
        - Used to create a matrix with the true/false predictions on the dev set
    - F1_score
        - Used to print the F1_score on the dev set for the 3 different types, macro, micro, and weighted
    - StandardScaler
        - Used to scale the data from the feature extraction
    - Pipeline
        - Used to put the data through the scaler before model training
- Numpy
    - Round
        - Just used when printing the accuracy/f1/recall scores
- Tensorflow
    - Keras.layers
        - InputLayer
            - Used to convert the manually defined features into the first layer of the MLP
        - Dense
            - Used as the inner hidden layers of the MLP
        - Dropout
            - Used between hidden layers to apply dropout effect
    - Keras.models
        - Sequential

            - Used to string together all the layers into one model
        - Keras.initializers
            - GlorotNormal
                - A version of the data initialization model discussed in class, where starting weights of model are chosen semi-randomly on a normal distribution from the number of input and output units in the layer
- Seaborn
    - Heatmap
        - Used to graphically display the confusion matrix generated by the sklearn function
- Keras
    - wrappers.scikit_learn.KerasClassifier
        - Used to convert the tensorflow keras model into an sklearn model that can be trained with the GridSearch function
- MatPlotLib
    - Pyplot
        - Used to add labels to the confusion matrix displayed

Experience and lessons

The improvement in terms of accuracy from the logistic regression model created in the midterm to the multi layer perceptron created here was very noticeable, and improvement of ~ 12-15 percent. One of the most helpful things that I changed when working on this project was using a python workbook with jupyter instead of programming in a normal python IDE like before. Since I had already worked out most of my syntax errors, I didn't need many of the features that pycharm, the IDE that I was using, provided. One of the biggest wastes of time in the midterm was that everytime I wanted to retest my model, I had to recreate my feature dataframe continuously. The benefit of using the workbook was that I could run chunks of my code at a time while still retaining the results of the other cell's runs. This way, I could run just the part of my code that worked on the model, saving minutes each time I did so, which added up greatly over the course of development. One of the more interesting things I learned is that a model being very deep does not necessarily make it better. I anticipated using at least 5 hidden layers before I actually started, but realized that it would be almost impossible to tune such a model without a much more powerful machine. It was a shock that my most effective model was the one with only one hidden layer, but after testing the alternatives manually, I feel that I have a greater intuitive sense of why this is the case. Overall, this project was a great window into the upsides and downsides of training a deep learning model, the pitfalls of trying to understand the reasoning behind model changes and accuracy, and the amount of time/resources necessary to train the massive models that companies like Google creates.