B455Project4

April 11, 2021

1 B455 Project 4

By Owen Gordon

For this project I am going to perform sentiment analysis on movie reviews from IMDb. Each review is given a rating of 1 to 10 stars. My models will try to predict the overall sentiment - positive if 6 stars or greater, and negative if 5 stars or fewer - and also try to predict the actual number of stars.

The first step in performing the analysis is to load the training and testing data. There are 50,000 total samples, 25,000 training samples and 25,000 testing samples. Within each group of training and testing samples, there are 12,500 positive reviews and 12,500 negative reviews.

I used the .open function in python to load the files, and then I split each line into a vector containing both the label (number of stars) and the words in the review. The file is in LIBSVM format which means that the data is stored {"word" : "frequency"}. The "word" is a number corresponding to the word index in the imbd.vocab file, and the "frequency" is the number of times that word appears in the review.

Note: The original files containing the training and testing data are both titled "labeledBow.feat", so I had to change the names to be able to contain both in the root directory. My file containing the training data is titled "train_labeledBow.feat" and my file containing the testing data is titled "test_labeledBow.feat".

[16]:
```
# read in the files
train = open('train_labeledBow.feat', 'r')
test = open('test_labeledBow.feat', 'r')

# read the lines in the file and split at each space to seperate the words
train_features = [line.split(' ') for line in train.readlines()]
test_features = [line.split(' ') for line in test.readlines()]
```

The final part in the preprocessing is to take the vector for each review, and extract the numerical values. Since the "word" is split from the "frequency" with a colon, I split each string into an array with two values, the first being the word number, and the second being the word frequency in the review. Then to aid in searching later on, I turned this array into a dictionary with all the keys being the word numbers, and the corresponding values being the frequency. This dictionary, along with the review target was placed into another dictionary with two keys. The first key is "output" and contains the integer value of the number of stars the review gave, and the second key is "input" which contains the dictionary of word - frequency pairs. The last step is to put each of

these dictionaries into a large array that contains all of the training data and one that contains all of the testing data.

The data should now be fully preprocessed and ready to be used in model training.

[17]:
```
# arrays that will contain all of the data
train_data = []
test_data = []

# loop over all of the training features and testing features
for train_feature, test_feature in zip(train_features, test_features):
  # split word-frequency for the inputs
  train_inputs = [f.split(':') for f in train_feature[1:]]
  test_inputs = [f.split(':') for f in test_feature[1:]]
  train_input_dict = {}
  test_input_dict = {}

  # create key-value pairs in the input dictionary
  for train in train_inputs:
      train_input_dict[int(train[0].strip('\n'))] = int(train[1].strip('\n'))
  for test in test_inputs:
    test_input_dict[int(test[0].strip('\n'))] = int(test[1].strip('\n'))

  # put output and intput data into dictionary
  train_d = {'output': int(train_feature[0]), 'input': train_input_dict}
  test_d = {'output': int(test_feature[0]), 'input': test_input_dict}

  # add dictionaries to arrays that contain all the data
  train_data.append(train_d)
  test_data.append(test_d)
```

For any of the inputs to make sense, I need assign word polarity. For that, I am going to use the word polarities given in the "imdbEr.txt" file. This file contains a value for every word in the review dictionaries. I applied the same technique to read this data in as I used to read in the other files.

[8]:
```
# open the file
imdbEr = open('imdbEr.txt', 'r')

# read the float value in each line
vocab_values = [float(v.strip('\n')) for v in imdbEr.readlines()]
```

The first models I am going to train will try to predict the exact number of stars as outputs. I am going to use sklearn for the model types. The sklearn models that I am going to use are LogisticRegression, MLPClassifier, and MLPRegressor. I am going to reduce each input down into a single value which corresponsds to the sum of all word polarities in the review. This will use just the word occurence, not frequency.

[28]:
```
# calculuate polarity values for each review
X_train_occurrence = []
X_test_occurrence = []

for train, test in zip(train_data, test_data):
  train_sum = 0
  test_sum = 0
  for word in train['input'].keys():
    train_sum += vocab_values[word]
  X_train_occurrence.append(train_sum)
  for word in test['input'].keys():
    test_sum += vocab_values[word]
  X_test_occurrence.append(test_sum)
```

[29]:
```
# collect targets into single array
y_train_multi = [train['output'] for train in train_data]
y_test_multi = [test['output'] for test in test_data]
```

[30]:
```
# reshape input arrays
import numpy as np
X_train_occurrence_np = np.array(X_train_occurrence).reshape(-1, 1)
X_test_occurrence_np = np.array(X_test_occurrence).reshape(-1, 1)
```

[32]:
```
# create the models and test accuracy using occurence
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier, MLPRegressor

logistic_multi_occurrence = LogisticRegression()
logistic_multi_occurrence.fit(X_train_occurrence_np, y_train_multi)
logistic_multi_occurrence_accuracy = logistic_multi_occurrence.
 ↪score(X_test_occurrence_np, y_test_multi)

mlp_multi_occurrence = MLPClassifier()
mlp_multi_occurrence.fit(X_train_occurrence_np, y_train_multi)
mlp_multi_occurrence_accuracy = mlp_multi_occurrence.
 ↪score(X_test_occurrence_np, y_test_multi)

mlpr_multi_occurrence = MLPRegressor()
mlpr_multi_occurrence.fit(X_train_occurrence_np, y_train_multi)
mlpr_multi_occurrence_accuracy = mlpr_multi_occurrence.
 ↪score(X_test_occurrence_np, y_test_multi)

print("Accuracy of multi class prediction using word occurrence")
print(f"Logistic Regression accuracy: {logistic_multi_occurrence_accuracy}")
print(f"MLP Classifier accuracy: {mlp_multi_occurrence_accuracy}")
print(f"MLP Regressor accuracy: {mlpr_multi_occurrence_accuracy}")
```

```
Accuracy of multi class prediction using word occurrence
Logistic Regression accuracy: 0.36648
MLP Classifier accuracy: 0.36312
MLP Regressor accuracy: 0.5715366909038879
```

Using multi class output, and word occurrence, it is clear that the MLP Regressor has the best prediction. 57% accurency is not phenomenal, but considering that there were 10 possible classes, and it was correct over half the time, this is not a terrible model. The other two models were very similar in their results. Both had a ~1/3 success rate. Given that a random guess would lead to 10% accurary, anything over 10% is learning.

The next models I am going to try is the multi class prediction using word frequency as well. I am going to reduce the inputs again, and this time multipy the polarity value by the frequency of that word in the review. Given that word occurence is usually more important than the frequency in the text, I anticipate that this will not dramatically improve the predictions.

[ ]:
```
# calculuate polarity values for each review - this time incorporating word␣
↪frequency
X_train_frequency = []
X_test_frequency = []

for train, test in zip(train_data, test_data):
  train_sum = 0
  test_sum = 0
  for word in train['input'].keys():
    train_sum += (vocab_values[word] * train['input'][word])
  X_train_frequency.append(train_sum)
  for word in test['input'].keys():
    test_sum += (vocab_values[word] * test['input'][word])
  X_test_frequency.append(test_sum)
```

[37]:
```
# reshape input arrays again
X_train_frequency_np = np.array(X_train_frequency).reshape(-1, 1)
X_test_frequency_np = np.array(X_test_frequency).reshape(-1, 1)
```

[38]:
```
# create the models and test accuracy using frequency
logistic_multi_frequency = LogisticRegression()
logistic_multi_frequency.fit(X_train_frequency_np, y_train_multi)
logistic_multi_frequency_accuracy = logistic_multi_frequency.
↪score(X_test_frequency_np, y_test_multi)

mlp_multi_frequency = MLPClassifier()
mlp_multi_frequency.fit(X_train_frequency_np, y_train_multi)
mlp_multi_frequency_accuracy = mlp_multi_frequency.score(X_test_frequency_np,␣
↪y_test_multi)

mlpr_multi_frequency = MLPRegressor()
mlpr_multi_frequency.fit(X_train_frequency_np, y_train_multi)
```

```
mlpr_multi_frequency_accuracy = mlpr_multi_frequency.score(X_test_frequency_np,␣
↪y_test_multi)

print("Accuracy of multi class prediction using word frequency")
print(f"Logistic Regression accuracy: {logistic_multi_frequency_accuracy}")
print(f"MLP Classifier accuracy: {mlp_multi_frequency_accuracy}")
print(f"MLP Regressor accuracy: {mlpr_multi_frequency_accuracy}")
```

```
Accuracy of multi class prediction using word frequency
Logistic Regression accuracy: 0.35984
MLP Classifier accuracy: 0.35656
MLP Regressor accuracy: 0.534893570602633
```

As expected the results are largely similar to the previous results. The MLP Rregressor had the best accuracy, and the other two models had similar accuracies. Also as expected, the results were slightly worse. This is proof that using word occurence is more powerful than using word frequency.

The final models I am going to train using this reduction method are models that train using the binary (positive/negative) classes. Since word occurrence seems to be better than frequency, I am going to use the polarity values that are only based on word occurrence.

[20]:
```
# collect targets into single array - using binary values
y_train_binary = [1 if train['output'] > 5 else 0 for train in train_data]
y_test_binary = [1 if test['output'] > 5 else 0 for test in test_data]
```

[42]:
```
# create the models and test accuracy using occurence and binary outputs
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier, MLPRegressor

logistic_binary_occurrence = LogisticRegression()
logistic_binary_occurrence.fit(X_train_occurrence_np, y_train_binary)
logistic_binary_occurrence_accuracy = logistic_binary_occurrence.
↪score(X_test_occurrence_np, y_test_binary)

mlp_binary_occurrence = MLPClassifier()
mlp_binary_occurrence.fit(X_train_occurrence_np, y_train_binary)
mlp_binary_occurrence_accuracy = mlp_binary_occurrence.
↪score(X_test_occurrence_np, y_test_binary)

mlpr_binary_occurrence = MLPRegressor()
mlpr_binary_occurrence.fit(X_train_occurrence_np, y_train_binary)
mlpr_binary_occurrence_accuracy = mlpr_binary_occurrence.
↪score(X_test_occurrence_np, y_test_binary)

print("Accuracy of binary class prediction using word occurrence")
print(f"Logistic Regression accuracy: {logistic_binary_occurrence_accuracy}")
print(f"MLP Classifier accuracy: {mlp_binary_occurrence_accuracy}")
print(f"MLP Regressor accuracy: {mlpr_binary_occurrence_accuracy}")
```

```
Accuracy of binary class prediction using word occurrence
Logistic Regression accuracy: 0.8498
MLP Classifier accuracy: 0.85048
MLP Regressor accuracy: 0.5468018166918556
```

Switching to a binary output dramtically improved the accuracy of the predictions. The Logistic Regression accuracy and MLP Classifier accuary are now both around 85%. Surprisingly, the MLP Regressor, which was the best predictor on mutliclass, is the worst predictor on binary class outputs.

It is clear that the prediction of binary outputs is easier, and more successful. The combination binary outputs, and using word occurrence over frequency has lead to the best results so far. For the next part of my report I am going to use a "Bag-of-words" model to try to improve the predictions.

2 Bag of Words Model

For the next part of my project I intend to use a bag of words technique where the inputs are large feature vectors, and each dimension of the vector is a different word from the review dictionary. But first, it is important to realize that there are ~90,000 words, and 50,000 samples. If each sample has an input vector with 90,000 dimensions, this would mean there would be 50000 x 90000 = 4.5 billion numbers. Since each number is a double, this would end up being 36 billion bytes of data, or about 36 gigabytes - which is too much memory. To get around this bottleneck I instead used minibatch training.

Minibatch training is where you train on small batches of the total input. This way, you use less memory, and still are able to train the model on all of the data. I chose a minibatch size of 1000 inputs. This means that each iteration of the partial fitting, the model only sees 1000 inputs, but over the course of 25 iterations the model will see every training sample. I also rotated through the testing samples in the same manner.

I intend to train two different models. The first is a SGDClassifer, which when the argument loss='log' is given to the model, acts like logistic regression during partial fitting. The second is another MLPClassifer. This was the best performing model when using the reduction method.

[19]:
```
# define minibatch size and number of iterations
minibatch_size = 1000
total_size = np.ceil(len(train_data))
iterations = int(total_size / minibatch_size)
```

[73]:
```
# create SGDClassifier
from sklearn.linear_model import SGDClassifier
clf = SGDClassifier(loss='log')
```

When I create the input vectors I am just going to use word occurrence in the review, not word frequency. If the word appears in the review, the feature vector will have the polarity value in that dimension, if it doesn't then just a zero.

[90]:
```
# train over number of iterations
for i in range(iterations):
  X_train_minibatch = []
```

```
  # grab training targets
  y_train_minibatch = y_train_binary[i * minibatch_size:(i + 1) *␣
↪minibatch_size]

  X_test_minibatch = []

  # grab testing targets
  y_test_minibatch = y_test_binary[i * minibatch_size:(i + 1) * minibatch_size]

  # generate feature vectors
  for train, test in zip(train_data[i * minibatch_size:(i + 1) *␣
↪minibatch_size], test_data[i * minibatch_size:(i + 1) * minibatch_size]):
    train_inputs = train['input']
    train_vector = []
    test_inputs = train['input']
    test_vector = []

    # look at all words and check if they exist in review words
    # if the word exists append the polarity value
    # if not, append 0
    for word in range(len(vocab_values)):
      if word in train_inputs:
        train_vector.append(vocab_values[word])
      else:
        train_vector.append(0)

      if word in test_inputs:
        test_vector.append(vocab_values[word])
      else:
        test_vector.append(0)

    # add vectors to the minibatch
    X_train_minibatch.append(train_vector)
    X_test_minibatch.append(test_vector)

  # test the fit after minibatches are created
  clf.partial_fit(X_train_minibatch, y_train_minibatch, [0, 1])
  validation_accuracy = clf.score(X_test_minibatch, y_test_minibatch)
  training_accuracy = clf.score(X_train_minibatch, y_train_minibatch)
  print(f"Iteration {i + 1}/{iterations}: training accuracy -␣
↪{training_accuracy}, validation accuracy - {validation_accuracy}")
```

```
Iteration 1/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 2/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 3/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 4/25: training accuracy - 1.0, validation accuracy - 1.0
```

```
Iteration 5/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 6/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 7/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 8/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 9/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 10/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 11/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 12/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 13/25: training accuracy - 0.923, validation accuracy - 0.923
Iteration 14/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 15/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 16/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 17/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 18/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 19/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 20/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 21/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 22/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 23/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 24/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 25/25: training accuracy - 1.0, validation accuracy - 1.0
```

[76]:
```python
# create MLPClassifier - using 1000 neurons in hidden layer

from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(1000)
```

[77]:
```python
# do same thing for mlp classifier

for i in range(iterations):
  X_train_minibatch = []
  y_train_minibatch = y_train_binary[i * minibatch_size:(i + 1) *␣
↪minibatch_size]

  X_test_minibatch = []
  y_test_minibatch = y_test_binary[i * minibatch_size:(i + 1) * minibatch_size]

  for train, test in zip(train_data[i * minibatch_size:(i + 1) *␣
↪minibatch_size], test_data[i * minibatch_size:(i + 1) * minibatch_size]):
    train_inputs = train['input']
    train_vector = []
    test_inputs = train['input']
    test_vector = []

    for word in range(len(vocab_values)):
      if word in train_inputs:
        train_vector.append(vocab_values[word])
```

```
      else:
        train_vector.append(0)

      if word in test_inputs:
        test_vector.append(vocab_values[word])
      else:
        test_vector.append(0)

    X_train_minibatch.append(train_vector)
    X_test_minibatch.append(test_vector)

  mlp._partial_fit(X_train_minibatch, y_train_minibatch, [0, 1])
  validation_accuracy = mlp.score(X_test_minibatch, y_test_minibatch)
  training_accuracy = mlp.score(X_train_minibatch, y_train_minibatch)
  print(f"Iteration {i + 1}/{iterations}: training accuracy -␣
↪{training_accuracy}, validation accuracy - {validation_accuracy}")
```

```
Iteration 1/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 2/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 3/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 4/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 5/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 6/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 7/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 8/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 9/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 10/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 11/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 12/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 13/25: training accuracy - 0.5, validation accuracy - 0.5
Iteration 14/25: training accuracy - 0.0, validation accuracy - 0.0
Iteration 15/25: training accuracy - 0.0, validation accuracy - 0.0
Iteration 16/25: training accuracy - 0.0, validation accuracy - 0.0
Iteration 17/25: training accuracy - 0.0, validation accuracy - 0.0
Iteration 18/25: training accuracy - 0.0, validation accuracy - 0.0
Iteration 19/25: training accuracy - 0.0, validation accuracy - 0.0
Iteration 20/25: training accuracy - 0.0, validation accuracy - 0.0
Iteration 21/25: training accuracy - 0.0, validation accuracy - 0.0
Iteration 22/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 23/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 24/25: training accuracy - 1.0, validation accuracy - 1.0
Iteration 25/25: training accuracy - 1.0, validation accuracy - 1.0
```

When you look at the results of this training, there is an obvious reaction, and a not so obvious reason. The obvious thing you see is that the validation accuracy is 100%. There is one reason this accuracy is so high, and its not because the model is perfect. It is because of the way the training samples were created: every postive sample come before every negative sample. The model is recognizing this and, weighting the appropriate output so heavily, that it always has the same

output. This is clearly not correct, so I am going to shuffle the inputs and see if the model can learn this way instead.

[21]:
```
import sklearn
train_data_shuffled = sklearn.utils.shuffle(train_data)
test_data_shuffled = sklearn.utils.shuffle(test_data)
```

[22]:
```
# collect shuffled targets
y_train_shuffled_binary = [1 if train['output'] > 5 else 0 for train in␣
↪train_data_shuffled]
y_test_shuffled_binary = [1 if test['output'] > 5 else 0 for test in␣
↪test_data_shuffled]
```

[25]:
```
# logistic regression learning with shuffled inputs
clf_shuffled = SGDClassifier(loss='log')

for i in range(iterations):
  X_train_minibatch = []
  y_train_minibatch = y_train_shuffled_binary[i * minibatch_size:(i + 1) *␣
↪minibatch_size]

  X_test_minibatch = []
  y_test_minibatch = y_test_shuffled_binary[i * minibatch_size:(i + 1) *␣
↪minibatch_size]

  for train, test in zip(train_data_shuffled[i * minibatch_size:(i + 1) *␣
↪minibatch_size], test_data_shuffled[i * minibatch_size:(i + 1) *␣
↪minibatch_size]):
    train_inputs = train['input']
    train_vector = []
    test_inputs = train['input']
    test_vector = []

    for word in range(len(vocab_values)):
      if word in train_inputs:
        train_vector.append(vocab_values[word])
      else:
        train_vector.append(0)

      if word in test_inputs:
        test_vector.append(vocab_values[word])
      else:
        test_vector.append(0)

    # add vectors to the minibatch
    X_train_minibatch.append(train_vector)
    X_test_minibatch.append(test_vector)
```

```
  # test the fit after minibatches are created
  clf_shuffled.partial_fit(X_train_minibatch, y_train_minibatch, [0, 1])
  validation_accuracy = clf_shuffled.score(X_test_minibatch, y_test_minibatch)
  training_accuracy = clf_shuffled.score(X_train_minibatch, y_train_minibatch)
  print(f"Iteration {i + 1}/{iterations}: training accuracy -␣
↪{training_accuracy}, validation accuracy - {validation_accuracy}")
```

```
Iteration 1/25: training accuracy - 0.945, validation accuracy - 0.49
Iteration 2/25: training accuracy - 0.876, validation accuracy - 0.514
Iteration 3/25: training accuracy - 0.948, validation accuracy - 0.492
Iteration 4/25: training accuracy - 0.951, validation accuracy - 0.492
Iteration 5/25: training accuracy - 0.964, validation accuracy - 0.515
Iteration 6/25: training accuracy - 0.947, validation accuracy - 0.478
Iteration 7/25: training accuracy - 0.954, validation accuracy - 0.529
Iteration 8/25: training accuracy - 0.958, validation accuracy - 0.498
Iteration 9/25: training accuracy - 0.941, validation accuracy - 0.498
Iteration 10/25: training accuracy - 0.935, validation accuracy - 0.516
Iteration 11/25: training accuracy - 0.96, validation accuracy - 0.509
Iteration 12/25: training accuracy - 0.923, validation accuracy - 0.478
Iteration 13/25: training accuracy - 0.945, validation accuracy - 0.508
Iteration 14/25: training accuracy - 0.946, validation accuracy - 0.479
Iteration 15/25: training accuracy - 0.945, validation accuracy - 0.466
Iteration 16/25: training accuracy - 0.944, validation accuracy - 0.475
Iteration 17/25: training accuracy - 0.901, validation accuracy - 0.542
Iteration 18/25: training accuracy - 0.966, validation accuracy - 0.508
Iteration 19/25: training accuracy - 0.965, validation accuracy - 0.506
Iteration 20/25: training accuracy - 0.961, validation accuracy - 0.525
Iteration 21/25: training accuracy - 0.956, validation accuracy - 0.47
Iteration 22/25: training accuracy - 0.967, validation accuracy - 0.481
Iteration 23/25: training accuracy - 0.977, validation accuracy - 0.519
Iteration 24/25: training accuracy - 0.964, validation accuracy - 0.5
Iteration 25/25: training accuracy - 0.966, validation accuracy - 0.489
```

[106]:
```
# mlp learning with shuffled inputs
mlp_shuffled = MLPClassifier(1000)

for i in range(iterations):
  X_train_minibatch = []
  y_train_minibatch = y_train_shuffled_binary[i * minibatch_size:(i + 1) *␣
↪minibatch_size]

  X_test_minibatch = []
  y_test_minibatch = y_test_shuffled_binary[i * minibatch_size:(i + 1) *␣
↪minibatch_size]
```

```
 for train, test in zip(train_data_shuffled[i * minibatch_size:(i + 1) *␣
↪minibatch_size], test_data_shuffled[i * minibatch_size:(i + 1) *␣
↪minibatch_size]):
   train_inputs = train['input']
   train_vector = []
   test_inputs = train['input']
   test_vector = []

   for word in range(len(vocab_values)):
     if word in train_inputs:
       train_vector.append(vocab_values[word])
     else:
       train_vector.append(0)

     if word in test_inputs:
       test_vector.append(vocab_values[word])
     else:
       test_vector.append(0)

   X_train_minibatch.append(train_vector)
   X_test_minibatch.append(test_vector)

 mlp_shuffled._partial_fit(X_train_minibatch, y_train_minibatch, [0, 1])
 validation_accuracy = mlp_shuffled.score(X_test_minibatch, y_test_minibatch)
 training_accuracy = mlp_shuffled.score(X_train_minibatch, y_train_minibatch)
 print(f"Iteration {i + 1}/{iterations}: training accuracy -␣
↪{training_accuracy}, validation accuracy - {validation_accuracy}")
```

```
Iteration 1/25: training accuracy - 0.965, validation accuracy - 0.505
Iteration 2/25: training accuracy - 0.947, validation accuracy - 0.508
Iteration 3/25: training accuracy - 0.936, validation accuracy - 0.521
Iteration 4/25: training accuracy - 0.934, validation accuracy - 0.502
Iteration 5/25: training accuracy - 0.934, validation accuracy - 0.51
Iteration 6/25: training accuracy - 0.932, validation accuracy - 0.521
Iteration 7/25: training accuracy - 0.926, validation accuracy - 0.482
Iteration 8/25: training accuracy - 0.943, validation accuracy - 0.506
Iteration 9/25: training accuracy - 0.943, validation accuracy - 0.492
Iteration 10/25: training accuracy - 0.942, validation accuracy - 0.515
Iteration 11/25: training accuracy - 0.938, validation accuracy - 0.498
Iteration 12/25: training accuracy - 0.952, validation accuracy - 0.502
Iteration 13/25: training accuracy - 0.945, validation accuracy - 0.497
Iteration 14/25: training accuracy - 0.943, validation accuracy - 0.506
Iteration 15/25: training accuracy - 0.933, validation accuracy - 0.491
Iteration 16/25: training accuracy - 0.934, validation accuracy - 0.512
Iteration 17/25: training accuracy - 0.933, validation accuracy - 0.502
Iteration 18/25: training accuracy - 0.943, validation accuracy - 0.52
Iteration 19/25: training accuracy - 0.93, validation accuracy - 0.49
```

```
Iteration 20/25: training accuracy - 0.937, validation accuracy - 0.515
Iteration 21/25: training accuracy - 0.925, validation accuracy - 0.518
Iteration 22/25: training accuracy - 0.939, validation accuracy - 0.506
Iteration 23/25: training accuracy - 0.933, validation accuracy - 0.522
Iteration 24/25: training accuracy - 0.928, validation accuracy - 0.516
Iteration 25/25: training accuracy - 0.934, validation accuracy - 0.506
```

After shuffling the inputs, the accuarcy hovered around 50%, not much better than random guess, so this method didn't work all too well. There is potential that more training could improve this method, but each of these training cycles took about 25-30 minutes, so it would have to potentially run for weeks to train enough to achieve a higher accuracy.

The last thing I am going to try in this project is do bag-of-words, but only on a certain number of the most common words, instead of the entire vocabuarly dictionary.

The first step to do this will be to rank each word by totaling the number of times that word appeared in the reviews. Again, I am only going to look at word occurrence in each review, not the total number of times a word was used. For words like "and" and "the" and "but" this will reduce their total popularity.

Since only the training inputs are used to fit the model, I am only going to look at the words in the training data.

[5]:
```
def get_n_popular_words(n):
  words = {}
  for word in range(len(vocab_values)):
    words[word] = 0

  # count the number of word occurences
  for train in train_data:
    inputs = train['input']
    for word in inputs:
      words[word] += 1

  sorted_words = sorted(words.items(), key=lambda item: item[1])
  n_sorted_words = sorted_words[-n:]
  most_popular = [p[0] for p in n_sorted_words]
  return most_popular
```

Now I can use feature vectors of any size, and hopefully fun the code faster to train more models. I am going to begin with the 1000 most popular words.

[6]:
```
def create_vector(input, most_popular):
  vector = []
  for word in most_popular:
    if word in input:
      vector.append(vocab_values[word])
    else:
      vector.append(0)
  return vector
```

[126]:
```
# logistic regression and mlp learning with 1000 most popular words
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier

n = 1000 # number of popular words

# creating mlp
mlp_1000_most_popular = MLPClassifier(1000)

#creating logistic regression
lr_1000_most_popular = LogisticRegression(max_iter=200)

X_train_1000 = []
X_test_1000 = []

most_popular_1000 = get_n_popular_words(n) # get most popular words

# generate vectors
for train, test in zip(train_data, test_data):
  X_train_1000.append(create_vector(train['input'], most_popular_1000))
  X_test_1000.append(create_vector(test['input'], most_popular_1000))

lr_1000_most_popular.fit(X_train_1000, y_train_binary)
lr_most_popular_1000_accuracy = lr_1000_most_popular.score(X_test_1000,␣
 ↪y_test_binary)

# training mlp
mlp_1000_most_popular.fit(X_train_1000, y_train_binary)
mlp_most_popular_1000_accuracy = mlp_1000_most_popular.score(X_test_1000,␣
 ↪y_test_binary)

print("Using 1000 most popular words")
print(f"Logistic Regression accuracy - {lr_most_popular_1000_accuracy}")
print(f"MLPClassifier accuracy - {mlp_most_popular_1000_accuracy}")
```

```
Using 1000 most popular words
Logistic Regression accuracy - 0.85864
MLPClassifier accuracy - 0.85568
```

Switching to the n most popular words is an extremely sizable improvement. Using just the first 1000 most popular words, the accuarcy of both Logistic Regression and the MLP Classifier are around 85%. This is much better, and these models are much more accurate than the previous bag-of-words models.

The final models I am going to train are the same model types (Logistic Regression, MLPClassifier), and this time use the 5000 most popular words.

[13]:
```
# logistic regression and mlp learning with 5000 most popular words
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier

n = 5000 # number of popular words

# creating mlp
mlp_5000_most_popular = MLPClassifier(1000)

#creating logistic regression
lr_5000_most_popular = LogisticRegression(max_iter=200)

X_train_5000 = []
X_test_5000 = []

most_popular_5000 = get_n_popular_words(n) # get most popular words

# generate vectors
for train, test in zip(train_data, test_data):
  X_train_5000.append(create_vector(train['input'], most_popular_5000))
  X_test_5000.append(create_vector(test['input'], most_popular_5000))

lr_5000_most_popular.fit(X_train_5000, y_train_binary)
lr_most_popular_5000_accuracy = lr_5000_most_popular.score(X_test_5000,␣
 ↪y_test_binary)

# training mlp
mlp_5000_most_popular.fit(X_train_5000, y_train_binary)
mlp_most_popular_5000_accuracy = mlp_5000_most_popular.score(X_test_5000,␣
 ↪y_test_binary)

print("Using 5000 most popular words")
print(f"Logistic Regression accuracy - {lr_most_popular_5000_accuracy}")
print(f"MLPClassifier accuracy - {mlp_most_popular_5000_accuracy}")
```

```
Using 5000 most popular words
Logistic Regression accuracy - 0.87464
MLPClassifier accuracy - 0.87412
```

3 Conclusion

In conclusion, I think that the models I trained were successful. The initial bag of words models trained in minibatch iterations were too large to train in any reasonable time, but the performance could improve given much more training. The models that simply reduced the vectors down were also very successful. These showed that word occurrence is better than word frequency which is what I used in the rest of my project. These models also showed that predicting overall sentiment - positive or negative - is significantly easier than the number of stars. Different people have different

criterion they use for their individual ratings, and it is more important to look at if the review was positive or negative, rather than the number of stars. The final models I trained, the ones that used sparse vectors of only the n most common words were also very successful. I think with more time I could fine tune the hyper-parameters of these models and achieve even better results.

I am impressed by how well these models performed, and I think they could all keep heading this direction to perform better and better after more training and time. The accuracy of the models increased from 1000 most popular to 5000 most popular. This increase shows that using more words improves the model's accuracy. The MLP model took far longer than the logistic regression model, while achieving the same accuracies. In the future, the MLP could be optimized, or logisitic regression could be used only.

Overall, I think that these models were successful at performing sentiment analysis on the IMDb dataset, and there are many optimizations that could improve this further.