How I chose my dataset for this assignment

When looking through the datasets on Kaggle.com, I tried to look for datasets that were preferably:

1. Relatively small (<= 5000 elements) or could feasibly be paired down to be smaller.
2. Had a relatively balanced distribution between two binary choices

After looking through, I decided to pick the "IMDB movie" dataset because it was relatively balanced, had simple to understand data that only consisted of a text field and a label, and was pretty balanced.

The original dataset had around 40k elements in it, so I edited the dataset itself to just get the first 5000 elements.

```
import pandas as pd
from nltk.corpus import stopwords
import nltk
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
import seaborn as sb
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
```

```
df = pd.read_csv('../content/data/movie.csv', header=0)
print('rows and columns:', df.shape)
print(df.head())
print(df.dtypes)
```

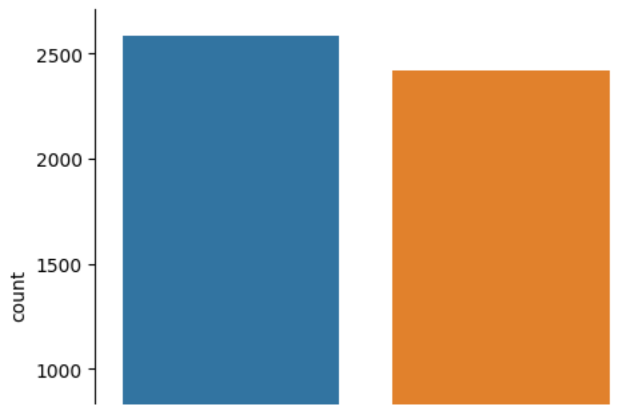
```
rows and columns: (5000, 2)
                                                text  label
0  I grew up (b. 1965) watching and loving the Th...      0
1  When I put this movie in my DVD player, and sa...      0
2  Why do people who do not know what a particula...      0
3  Even though I have great interest in Biblical ...      0
4  Im a die hard Dads Army fan and nothing will e...      1
text     object
label     int64
dtype: object
```

```
vectorizer = TfidfVectorizer()

X = df.text
y = df.label

# print to verify that I grabbed the right data
print(X.head())

# split text and targets into training data and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.8, random_state=22)

# check how much training data I have
print(X_train.shape)
```

```
0    I grew up (b. 1965) watching and loving the Th...
1    When I put this movie in my DVD player, and sa...
2    Why do people who do not know what a particula...
3    Even though I have great interest in Biblical ...
4    Im a die hard Dads Army fan and nothing will e...
Name: text, dtype: object
(4000,)
```

```
# plot the distribution of the datasets
sb.catplot(x="label", kind="count", data=df)
```

```
<seaborn.axisgrid.FacetGrid at 0x7fdba53d49a0>
```



Dataset Description

This dataset holds data for 5000 movie reviews. There are two columns:

- text - gives the text of a movie review from IMDB.
- label - states whether the review is positive or negative. 0 for negative, 1 for positive

Given the data, I want to be able to predict whether a review is positive or negative given the review itself.

I decided to keep the test size as 0.2, since I feel that 1000 test values would be a good indicator of how well the program was able to classify the data.

As mentioned earlier, I reduced the original size of the dataset to get the first 5000 items. I then checked to see whether they were relatively balanced, which they were.

Naive Bayes approach on the Dataset (Attempt 1)

I tried the BernoulliNB first since I thought that the presence or absence of certain words may indicate a positive or negative review (for instance, negative words like "hate" or "despise" may indicate a negative review. Though, not always since a review could still have those words and yet overall be positive).

I also decided to not use stopwords for my vectorizer.

```
# Vectorize the data
X_train = vectorizer.fit_transform(X_train)
X_test = vectorizer.transform(X_test)

print('train size:', X_train.shape)
print(X_train.toarray()[:5])

print('\ntest size:', X_test.shape)
print(X_test.toarray()[:5])
```

```
train size: (4000, 35081)
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]

test size: (1000, 35081)
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

```
from sklearn.naive_bayes import BernoulliNB
naive_bayes = BernoulliNB()
naive_bayes.fit(X_train, y_train)
```

```
▾ BernoulliNB
BernoulliNB()
```

```
# predict the test values given that we have trained our model already
pred = naive_bayes.predict(X_test)
```

```
print(confusion_matrix(y_test, pred))
```

```
[[483  49]
 [124 344]]
```

```
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
```

```
accuracy score:  0.827
precision score:  0.8753180661577609
recall score:  0.7350427350427351
f1 score:  0.7990708478513356
```

According to the data, the model was about 82% accurate.

In the test data, there are 2583 negative reviews from the 5000 total reviews. That makes it 51% of all the reviews. Therefore, we can say with certainty that my model did learn, as my model was able to score significantly higher than 51%.

I notice that there are a significant amount of false negatives in my confusion matrix: about 1/10th of the 1000 items in the test data were placed as a false negative. Let me analyze in the questions that my model got incorrect, with a particular focus on the false negatives.

```
print('negative reviews in test data:',y_test[y_test==0].shape[0])
print('test size: ', len(y_test))
```

```
negative reviews in test data: 532
test size:  1000
```

```
y_test[y_test != pred]
```

```
3408    1
4880    1
2981    1
4564    1
3851    1
       ..
3414    0
974     0
902     1
4480    1
964     0
Name: label, Length: 173, dtype: int64
```

```
for i in [3408, 4880, 2981, 4564, 3851, 3414, 974, 902, 4480, 964]:
  print(df.loc[i])
```

```
text     In a lot of ways this film defines the essence...
label                                                    1
Name: 3408, dtype: object
text     i can't believe people are giving bad reviews ...
label                                                    1
Name: 4880, dtype: object
text     I have never seen such a movie before. I was o...
label                                                    1
Name: 2981, dtype: object
text     I haven't written a review on IMDb for the lon...
label                                                    1
Name: 4564, dtype: object
text     Just a short comment! I want to say that I lik...
label                                                    1
Name: 3851, dtype: object
text     Bugs Bunny accidentally ends up at the South P...
label                                                    0
Name: 3414, dtype: object
text     HAPPY DAYS was one of my favorite shows when i...
label                                                    0
Name: 974, dtype: object
text     Didn't Mystic Pizza win the Oscar for that yea...
label                                                    1
Name: 902, dtype: object
text     It's depressing to see where Jackie Chan has e...
label                                                    1
```

```
Name: 4480, dtype: object
text     I guess I was prepared after all the years of ...
label                                                   0
Name: 964, dtype: object
```

I notice for a lot of these good reviews that my model classified as bad, they start with a lot of negative words like "depressing", "can't", and for the Mystic Pizza review uses a lot of words like "disturbed", "never had a chance", or "far-fetched" sprinkled throughout, but overall gave it a positive review, saying at the end that it is a "must-see".

I also noticed that some of the false negative reviews used a lot of exclamation points and ellipses, which may have given the impression of a bad review when it really wasn't.

Naive Bayes approach on the Dataset (Attempt 2)

I think maybe using n-grams could help to give a bit more context for whenever a negative word is used, so that my model can maybe find negative phrases better rather than just a negative word. This won't solve every issue (note the particular case of the Mystic Pizza review), but I believe this will help for those reviews that may just use negative words but would overall be positive. Luckily, TfidfVectorizer provides a way to utilize ngrams.

In addition, I think if I preprocess the text to replace punctuation with "punct" it may help to remove those false negatives that used frequent punctuation. I also want to remove anytime the text has the html for two line breaks as that is not really relevant info

```
from pandas._libs.tslibs import dtypes
import re
from nltk import ngrams

# replace unnecessary html in the data
df['text'].replace('<br /><br />', '', regex=True, inplace=True)

# replacing excessive punctuation with "punct"
df['text'].replace('[!@#.*][!@#.*]+', ' punct ', regex=True, inplace=True)


# providing unigrams and bigrams to the vectorizer
vectorizer = TfidfVectorizer(ngram_range=(1,2))

# checking out some known problem messages that had a lot of excessive punctuation
for i in [4880, 3851]:
  print(df.loc[i]['text'])
```

```
esn't follow the book punct  for sure punct the book by dean koontz is much better punct  but the movie is also good as well punct  it h
ginning until the end! I have it on tape and I can watch it a 100 times, it doesn't matter punct
```

```
# rerun the model again with the new data and vectorizer
X = df.text
y = df.label

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.8, random_state=22)

X_train = vectorizer.fit_transform(X_train)
X_test = vectorizer.transform(X_test)

naive_bayes.fit(X_train, y_train)
```

```
▾ BernoulliNB
BernoulliNB()
```

```
# evaluate

pred = naive_bayes.predict(X_test)
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
confusion_matrix(y_test, pred)
```

```
accuracy score:  0.753
precision score:  0.9233716475095786
recall score:  0.5149572649572649
```

```
f1 score:  0.6611796982167353
array([[512,  20],
       [227, 241]])
```

The resulting model seemed to do better at identifying the positive reviews, but did worse in determining which reviews were actually negative, which resulted in an overall worse performance. This may have been because the punctuation was actually used more often than not to determine a negative review, and taking that information away provided worse results.

Naive Bayes approach on the Dataset (Attempt 3)

For my final attempt, I will see if trigrams works better, or if setting the binary value to true in the vectorizer would help.

```
# completely refresh any preprocessing on df
df = pd.read_csv('../content/data/movie.csv', header=0)

# replace unnecessary html in the data
df['text'].replace('<br /><br />', '', regex=True, inplace=True)

# providing unigrams and trigrams to the vectorizer
vectorizer = TfidfVectorizer(ngram_range=(1,3), binary=True)

# running the model again
X = df.text
y = df.label

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.8, random_state=22)

X_train = vectorizer.fit_transform(X_train)
X_test = vectorizer.transform(X_test)

naive_bayes.fit(X_train, y_train)
```

```
▾ BernoulliNB
BernoulliNB()
```

```
pred = naive_bayes.predict(X_test)
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
confusion_matrix(y_test, pred)
```

```
accuracy score:  0.667
precision score:  0.9470198675496688
recall score:  0.3055555555555556
f1 score:  0.46203554119547663
array([[524,   8],
       [325, 143]])
```

it seems like the first attempt with minimal preprocessing performed the best. Precision increased a bit, but recall greatly decreased with the use of higher ngrams.

While the positives were able to be picked out even better, it seems like the negatives still had issues with being identified, resulting in the lowest score so far.

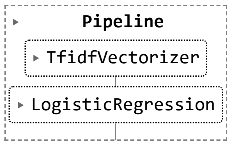
I believe using ngrams may have been the result of this, as it may have seen negative words and phrases and taken those to mean the review is negative when it was really a positive review.

Logistic Regression Attempt 1

I am going to do regression with the default settings and the lbfgs solver. I will also set up a pipeline so that I could modify parameters more easily on the next attempt.

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
```

```
df = pd.read_csv('../content/data/movie.csv', header=0)
X = df.text
y = df.label

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.8, random_state=22)

pipe = Pipeline([
        ('tfidf', TfidfVectorizer(binary=True)),
        ('logreg', LogisticRegression(solver='lbfgs', class_weight='balanced')),
])

pipe.fit(X_train, y_train)
```



```
pred = pipe.predict(X_test)
print(confusion_matrix(y_test, pred))
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
probs = pipe.predict_proba(X_test)
```

```
[[451  81]
 [ 52 416]]
accuracy score:  0.867
precision score:  0.8370221327967807
recall score:  0.8888888888888888
f1 score:  0.8621761658031087
```

Note that the initial attempt for logistic regression performed slighly better (86% accuracy) than the initial attempt for naive bayes (82% accuracy). This may have been because of the smaller bias on the data that logistic regression has when compared to the bias for naive bayes. It also may be because logistic regression is a discriminative classifier, which means it will be able to determine the most useful characteristics of positive and negative reviews and use that in determining what other reviews should be.

```
pipe.get_params()
```

```
{'memory': None,
 'steps': [('tfidf', TfidfVectorizer(binary=True)),
  ('logreg', LogisticRegression(class_weight='balanced'))],
 'verbose': False,
 'tfidf': TfidfVectorizer(binary=True),
 'logreg': LogisticRegression(class_weight='balanced'),
 'tfidf__analyzer': 'word',
 'tfidf__binary': True,
 'tfidf__decode_error': 'strict',
 'tfidf__dtype': numpy.float64,
 'tfidf__encoding': 'utf-8',
 'tfidf__input': 'content',
 'tfidf__lowercase': True,
 'tfidf__max_df': 1.0,
 'tfidf__max_features': None,
 'tfidf__min_df': 1,
 'tfidf__ngram_range': (1, 1),
 'tfidf__norm': 'l2',
 'tfidf__preprocessor': None,
 'tfidf__smooth_idf': True,
 'tfidf__stop_words': None,
 'tfidf__strip_accents': None,
 'tfidf__sublinear_tf': False,
 'tfidf__token_pattern': '(?u)\\b\\w\\w+\\b',
 'tfidf__tokenizer': None,
 'tfidf__use_idf': True,
 'tfidf__vocabulary': None,
 'logreg__C': 1.0,
 'logreg__class_weight': 'balanced',
 'logreg__dual': False,
 'logreg__fit_intercept': True,
 'logreg__intercept_scaling': 1,
 'logreg__l1_ratio': None,
 'logreg__max_iter': 100,
 'logreg__multi_class': 'auto',
 'logreg__n_jobs': None,
 'logreg__penalty': 'l2',
 'logreg__random_state': None,
 'logreg__solver': 'lbfgs',
 'logreg__tol': 0.0001,
 'logreg__verbose': 0,
 'logreg__warm_start': False}
```

Logistic Regression Attempt 2

I want to experiment with changing the penalty to L1 instead of the default L2. L1 is stated to result in a sparser model and helps the data points to converge more to a central point. I want to see what kind of effect this would have on a data set as small as this one. Since lbfgs is used for multiclass problems and I have a binary problem, I will experiment with changing the solver to be liblinear as that one is stated to be better for smaller datasets. Finally, I will increase the C value by just a bit to reduce any overfitting

```
pipe.set_params(logreg__penalty='l1', logreg__solver='liblinear', logreg__C=3.0).fit(X_train, y_train)

# see results
pred = pipe.predict(X_test)
print(confusion_matrix(y_test, pred))
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
probs = pipe.predict_proba(X_test)

pipe.set_params(logreg__penalty='l2', logreg__solver='liblinear', logreg__C=3.0).fit(X_train, y_train)

# see results
pred = pipe.predict(X_test)
print(confusion_matrix(y_test, pred))
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
probs = pipe.predict_proba(X_test)

pipe.set_params(logreg__penalty='l2', logreg__solver='lbfgs', logreg__C=3.0).fit(X_train, y_train)

# see results
pred = pipe.predict(X_test)
print(confusion_matrix(y_test, pred))
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
probs = pipe.predict_proba(X_test)
```

```
[[437  95]
 [ 54 414]]
accuracy score:  0.851
precision score:  0.8133595284872298
recall score:  0.8846153846153846
f1 score:  0.8474923234390992
[[453  79]
 [ 51 417]]
accuracy score:  0.87
precision score:  0.8407258064516129
recall score:  0.8910256410256411
f1 score:  0.8651452282157677
[[453  79]
 [ 51 417]]
accuracy score:  0.87
precision score:  0.8407258064516129
recall score:  0.8910256410256411
f1 score:  0.8651452282157677
```

Based on the results, it seems like using an l1 penalty made my model perform slightly worse than the base settings. In addition, changing the solver seemed to have no difference in terms of performance. Of the 3 parameters I modified, C was the one that made the difference. When I increased C to 3, the accuracy increased slightly. This may imply that my model was overfitting just a bit beforehand, which slightly reduced its accuracy score. If I increased C to any more than 3, it didn't make much of a difference.

Neural Networks Attempt 1

For this attempt, I will be using neural networks and see if its performance is better than the previous 2 approaches I have used. I will keep the initial steps the same in order to compare with the other models.

For the parameters specific to neural networks, I believe that there are 2 nodes in the input layer corresponding to the 2 columns of my dataset. Therefore, I think there should be 4 nodes in the hidden layer. I will start with a single hidden layer and see how it performs. I will keep the solver as lbfgs to be consistent with the logistic regression model.

```
df = pd.read_csv('../content/data/movie.csv', header=0)

vectorizer = TfidfVectorizer(binary=True)

X = vectorizer.fit_transform(df.text)
y = df.label

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.8, random_state=22)
```

```
from sklearn.neural_network import MLPClassifier

# training the model
classifier = MLPClassifier(solver='lbfgs', alpha=1e-5,
                   hidden_layer_sizes=(4,), random_state=1)
classifier.fit(X_train, y_train)
```

```
                    MLPClassifier
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(4,), random_state=1,
              solver='lbfgs')
```

```
pred = classifier.predict(X_test)
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
```

```
accuracy score:  0.865
precision score:  0.8336673346693386
recall score:  0.8888888888888888
f1 score:  0.860392967942089
```

The accuracy of the neural network is about on par with the best accuracy of the logistic regression model, and better than the best performance I was able to get for the naive bayes model. It seems like one hidden layer worked fairly well.

Neural Networks Attempt 2

In this attempt, I will go ahead and try to add the stopwords to the vectorizer, as that is something I had decided not to do in earlier models which may increase the performance in general.

```
from nltk.corpus import stopwords
import nltk
nltk.download('stopwords')

stopwords = list(set(stopwords.words('english')))
vectorizer = TfidfVectorizer(stop_words=stopwords, binary=True)

X = vectorizer.fit_transform(df.text)
y = df.label

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.8, random_state=22)
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
from sklearn.neural_network import MLPClassifier

# training the model
classifier = MLPClassifier(solver='lbfgs', alpha=1e-5,
                   hidden_layer_sizes=(4,), random_state=1)
classifier.fit(X_train, y_train)
```

```
▾                            MLPClassifier
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(4,), random_state=1,
              solver='lbfgs')
```

```
pred = classifier.predict(X_test)
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
```

```
accuracy score:  0.865
precision score:  0.8284023668639053
recall score:  0.8974358974358975
f1 score:  0.8615384615384615
```

Based on these results, it seems like taking into account stopwords made very little difference in the performance of this particular model. Perhaps stopwords are not a defining factor in determining whether a review is positive or negative.

Neural Networks Attempt 3

In this attempt, I want to increase the number of hidden layers and slightly increase the number of nodes in the hidden layer to determine if it would help the performance or if it would result in an overfit model.

In the first part, I will slightly increase the nodes in the hidden layer and check on its performance. In the second part, depending on how the first part goes (whether adding more nodes results in a better model), I will split either the 4 nodes or the slightly increased nodes into 2 hidden layers and check on performance.

```
# checking to see if increasing the nodes in the hidden layer will increase performance
classifier = MLPClassifier(solver='lbfgs', alpha=1e-5,
                   hidden_layer_sizes=(6,), random_state=1)
classifier.fit(X_train, y_train)

pred = classifier.predict(X_test)
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))

# checking if increasing the nodes even more would increase performance
classifier = MLPClassifier(solver='lbfgs', alpha=1e-5,
                   hidden_layer_sizes=(8,), random_state=1)
classifier.fit(X_train, y_train)

pred = classifier.predict(X_test)
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
```

```
accuracy score:  0.87
precision score:  0.8353174603174603
recall score:  0.8995726495726496
f1 score:  0.8662551440329218
accuracy score:  0.849
precision score:  0.8241308793456033
recall score:  0.8611111111111112
f1 score:  0.8422152560083594
```

Based on these results, 6 hidden nodes seems to be a good number in order to maximize the accuracy of the model.

```
# checking to see if splitting 6 nodes across 2 layers will increase performance
classifier = MLPClassifier(solver='lbfgs', alpha=1e-5,
                   hidden_layer_sizes=(3,3), random_state=1)
classifier.fit(X_train, y_train)

pred = classifier.predict(X_test)
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))

# checking another split of nodes
classifier = MLPClassifier(solver='lbfgs', alpha=1e-5,
                   hidden_layer_sizes=(4,2), random_state=1)
classifier.fit(X_train, y_train)

pred = classifier.predict(X_test)
print('accuracy score: ', accuracy_score(y_test, pred))
print('precision score: ', precision_score(y_test, pred))
print('recall score: ', recall_score(y_test, pred))
print('f1 score: ', f1_score(y_test, pred))
```

```
accuracy score:  0.864
precision score:  0.8387755102040816
recall score:  0.8782051282051282
f1 score:  0.8580375782881001
accuracy score:  0.532
precision score:  0.0
recall score:  0.0
f1 score:  0.0
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision is ill-defined and be
  _warn_prf(average, modifier, msg_start, len(result))
```

Based on these results, splitting the nodes across different layers does not have any significant difference. So for this particular dataset, a single hidden layer of 6 nodes would work well.

Conclusion

I believe in order to get the performance to be any better for any of these models, I would need to further analyze the data to find trends between positive and negative reviews. I would also need to further preprocess the data in different ways so that the model will be able to best determine what a review is. Another thing that I could do would be to bring back some more of the data that I had excluded from the original dataset, as having more data would allow for the model to train more.

✓ 5s completed at 8:05 PM