IST 736 Homework 7

Compare MNB and SVMs for Kaggle Sentiment Classificatio

IST 736 - Text Mining - Syracuse University

November 2022

In this assignment, I will be doing a comparison of Multinomial Bayes (MNB) and Support Vector Machine (SVM) models on a Rotten Tomatoes movie review dataset from Kaggle. There are three tasks; (1) training the models using unigram vectors, (2) training the models using unigram and bigram vectors, and (3) tuning the models and making a Kaggle submission.

I will start off by importing the necessary packages. All of the machine learning will be done through the Scikit-Learn package.

```
# Import standard packages
import pandas
import numpy

# Import packages from scikit-learn
from sklearn.pipeline import Pipeline
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
```

Next, I will import the data. The data comes from Kaggle and can be downloaded from the following web page https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews.

```
# Import data
data_train = pandas.read_csv("train.tsv", delimiter = "\t")
data_test = pandas.read_csv("test.tsv", delimiter = "\t")
X = data_train.Phrase.copy()
y = data_train.Sentiment.copy()
```

Next, I will split the data into train and test sets. I will be splitting the data into 60% train data and 40% test data. The same train and test sets will be used for all of the tasks in this report.

```
# Split data into test/train
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4, random_state = 42)
```

Let's take a look at the class balance. The majority vote on the test data is 31,597 "neutral" reviews, which is just a little over 50% of the observations.

```
# Take a look at the class balances
print("Train Class Balance", "\n", y_train.value_counts(), "\n")
print("Test Class Balance", "\n", y_test.value_counts())
```

```
Train Class Balance
 2    47985
3    19608
1    16348
4     5448
0     4247
```

```
Test Class Balance
 2    31597
3    13319
1    10925
4     3758
0     2825
```

Task 1 - Unigram Vectors

In the first task, I will be comparing MNB and SVM model using unigram vectors. The comparison will include the accuracy score, precision and recall scores, classification report, and top 10 most indicative words for "very negative" and "very positive" reviews. I will conclude the task with my interpretation of the results.

I will start off by vectorizing the train and test data. I will use the absolute frequency as the vectorization method for this task. Stopwords will be removed. The vectorizer is fit on the train data, and then the test data is vectorized using the fitted vectorizer.

```
# Initiate vectorizer
unigram_count_vectorizer = CountVectorizer(
  encoding = "latin-1",
  binary = False,
  min_df = 5,
  stop_words = "english"
)

# Fit the vectorizer on the train data and vectorize the train data
```

```
X_train_vec = unigram_count_vectorizer.fit_transform(X_train)

# Vectorize the test data
X_test_vec = unigram_count_vectorizer.transform(X_test)
```

Next, I will train the MNB and SVM models. For this task, the default arguments are used. After the models are trained, I will print out the accuracy score of the predictions it makes on the test data.

```
# Train a MNB classifier
nb_clf = MultinomialNB()
nb_clf.fit(X_train_vec, y_train)
print(nb_clf.score(X_test_vec,y_test))
```

```
0.6040625400486992
```

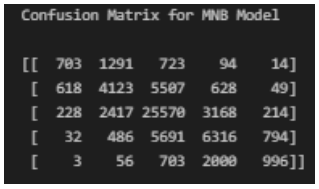
```
# Train a SVM classifier
svm_clf = LinearSVC(C = 1)
svm_clf.fit(X_train_vec, y_train)
print(svm_clf.score(X_test_vec, y_test))
```

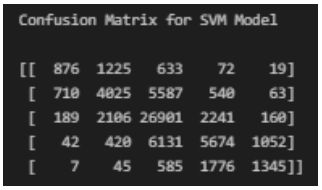
```
0.6218922209406639
```

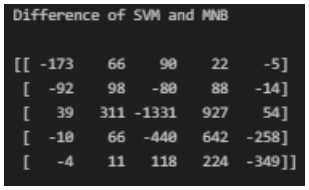
Both models were better than the majority vote. The SVM model outperformed the MNB model by about 2% accuracy. The MNB model correctly classified about 37,708 / 62,424 observations, while the SVM model correctly classified about 38,821 / 62,424 observations. The SVM model correctly classified about 1,113 more observations than the MNB model.

Next I will take a look at the confusion matrices.

```
# Print the confusion matrix for mnb and svm
print("Confusion Matrix for MNB Model", "\n")
print(confusion_matrix(y_test, nb_clf.predict(X_test_vec), labels = [0,1,2,3,4]), "\n")
print("Confusion Matrix for SVM Model", "\n")
print(confusion_matrix(y_test, svm_clf.predict(X_test_vec), labels = [0,1,2,3,4]), "\n")
print("Difference of SVM and MNB Confusion Matrix", "\n")
print(confusion_matrix(y_test, nb_clf.predict(X_test_vec), labels = [0,1,2,3,4]), "\n" -
      confusion_matrix(y_test, svm_clf.predict(X_test_vec), labels = [0,1,2,3,4]), "\n")
```







The confusion matrix for the MNB and SVM models are shown, along with another confusion matrix which takes MNB - SVM to highlight the differences. Interestingly, even though the SVM had a greater overall accuracy score, there were some areas where the MNB model had more correct classifications. The MNB model did slightly better on predicting "negative" reviews (label = 1), and also did better on predicting "positive" reviews (label = 3). The SVM model had the upper hand on predicting "very negative" reviews (label = 0), "neutral" reviews (label = 2), and "very positive" reviews (label = 4). It seems like the SVM model had a tendency to choose "very positive" more often than "positive", while the MNB model had a tendency to choose more "positive" more often than "very positive".

Next, I will look at the precision, recall, and classification reports.

```
# Print the precision, recall, and classification report for mnb
print("Precision for MNB Model")
print(precision_score(y_test, nb_clf.predict(X_test_vec), average = None), "\n")
print("Recall for MNB Model")
print(recall_score(y_test, nb_clf.predict(X_test_vec), average = None), "\n")
print("Classification Report for MNB Model")
print(classification_report(y_test, nb_clf.predict(X_test_vec), target_names = ["0","1","2","3","4"]), "\n")
```

```
# Print the precision, recall, and classification report for svm
print("Precision for SVM Model")
print(precision_score(y_test, svm_clf.predict(X_test_vec), average = None), "\n")
print("Recall for SVM Model")
print(recall_score(y_test, svm_clf.predict(X_test_vec), average = None), "\n")
print("Classification Report for SVM Model")
print(classification_report(y_test, svm_clf.predict(X_test_vec), target_names = ["0","1","2","3","4"]), "\n")
```

```
Precision for MNB Model
[0.44381313 0.4924161  0.66947688 0.51745043 0.48185776

Recall for MNB Model
[0.24884956 0.3773913  0.80925404 0.47420978 0.26503459

Classification Report for MNB Model
              precision    recall  f1-score   support

           0       0.44      0.25      0.32      2825
           1       0.49      0.38      0.43     10925
           2       0.67      0.81      0.73     31597
           3       0.52      0.47      0.49     13319
           4       0.48      0.27      0.34      3758

    accuracy                           0.60     62424
   macro avg       0.52      0.43      0.46     62424
weighted avg       0.58      0.60      0.59     62424
```

```
Precision for SVM Model
[0.48026316 0.51464007 0.67527675 0.55071338 0.50966275

Recall for SVM Model
[0.3100885  0.36842105 0.8513783  0.42600796 0.35790314

Classification Report for SVM Model
              precision    recall  f1-score   support

           0       0.48      0.31      0.38      2825
           1       0.51      0.37      0.43     10925
           2       0.68      0.85      0.75     31597
           3       0.55      0.43      0.48     13319
           4       0.51      0.36      0.42      3758

    accuracy                           0.62     62424
   macro avg       0.55      0.46      0.49     62424
weighted avg       0.60      0.62      0.60     62424
```

The SVM model had a better precision score than the MNB model for all labels. It had a better recall score for "very negative", "neutral", and "very positive", but the MNB model had a better recall score on "negative" and "positive". This aligns with what I was saying before because if the SVM model had a tendency to pick "very positive" over "positive", then it makes sense that it would have a better recall score in the "very positive" category and a worse recall score in the "positive category".

Lastly, I will look at the top 10 most indicative words for the "very positive" and "very negative" categories for both the MNB and SVM models.

```
# Save the feature ranks
feature_ranks_very_neg_mnb = sorted(zip(nb_clf.feature_log_prob_[0], unigram_count_vectorizer.get_feature_names()))
feature_ranks_very_pos_mnb = sorted(zip(nb_clf.feature_log_prob_[4], unigram_count_vectorizer.get_feature_names()))
feature_ranks_very_neg_svm = sorted(zip(svm_clf.coef_[0], unigram_count_vectorizer.get_feature_names()))
feature_ranks_very_pos_svm = sorted(zip(svm_clf.coef_[4], unigram_count_vectorizer.get_feature_names()))

# Top 10 very negative for mnb and svm
print("Top 10 Indicative Words for Very Negative Category MNB Model", "\n")
print(feature_ranks_very_neg_mnb[-10: ], "\n")
print("Top 10 Indicative Words for Very Negative Category SVM Model", "\n")
print(feature_ranks_very_neg_svm[-10: ])

# Top 10 very positive for mnb and svm
print("Top 10 Indicative Words for Very Positive Category MNB Model", "\n")
print(feature_ranks_very_pos_mnb[-10: ], "\n")
print("Top 10 Indicative Words for Very Positive Category SVM Model", "\n")
print(feature_ranks_very_pos_svm[-10: ])
```

Top 10 "very negative"

```
Top 10 Indicative Words for Very Negative Category MNB Model

[(-5.998191867857409, 'does'), (-5.998191867857409, 'time'), (-5.987142031670825, 'comedy'),
(-5.965402045034418, 'minutes'), (-5.8632725499580385, 'characters'), (-5.615199615601304,
'just'), (-5.204696713463182, 'like'), (-4.873984032000336, 'bad'), (-4.8246782710161815,
'film'), (-4.341022552602119, 'movie')]

Top 10 Indicative Words for Very Negative Category SVM Model

[(1.5336164509764454, 'awfulness'), (1.5405297690897386, 'dehumanizing'), (1.54337302776449,
'disappointment'), (1.5758202403840431, 'turd'), (1.600033955731724, 'worthless'),
(1.6015544590919197, 'repulsive'), (1.6573642235629853, 'unappealing'), (1.7640481155392784,
'unbearable'), (1.8708641989569756, 'dud'), (2.0389986088479817, 'unwatchable')]
```

Top 10 "very positive"

```
Top 10 Indicative Words for Very Positive Category MNB Model

[(-5.822900380967689, 'performance'), (-5.782734339242355, 'great'), (-5.706940499872821,
'performances'), (-5.706940499872821, 'story'), (-5.61628613160469, 'comedy'),
(-5.390479462870997, 'good'), (-5.270335151028934, 'funny'), (-5.150457873769435, 'best'),
(-4.809810265737742, 'movie'), (-4.286562121973194, 'film')]

Top 10 Indicative Words for Very Positive Category SVM Model

[(1.5381361274733105, 'glorious'), (1.5723646977021748, 'proud'), (1.6261954125662876,
'praiseworthy'), (1.6394560935074933, 'standout'), (1.7019431942483085, 'zings'),
(1.7518054168669333, 'luminous'), (1.7856525103165528, 'astoundingly'), (1.8370937987906062,
'perfection'), (1.892206189431805, 'celebrated'), (1.9519104101945723, 'refreshes')]
```

It is interesting that the words that the models learned are quite different. In general, I would argue that the words that the SVM model learned make more sense than the words that the MNB model learned. For example, the MNB model learned that the words "does", "time", "comedy", "minutes" are associated with "very negative" reviews. Those seem more like they should be neutral words. Meanwhile, the SVM model learned that "awfulness", "dehumanizing", "disappointment", "worthless" are associated with "very negative" reviews. That seems more realistic.

Task 2 - Unigram and Bigram Vectors

In this task, I will repeat all of the same steps as the previous task, except this time I will include bigram vectors in the vectorization process.

```
# Initiate vectorizer
gram12_count_vectorizer = CountVectorizer(
  encoding = "latin-1",
  ngram_range = (1,2),
  min_df = 5,
  stop_words = "english"
```

```
)

# Fit the vectorizer on the train data and vectorize the train data
X_train_vec = gram12_count_vectorizer.fit_transform(X_train)

# Vectorize the test data
X_test_vec = gram12_count_vectorizer.transform(X_test)
```

```
# Train a MNB classifier
nb_clf = MultinomialNB()
nb_clf.fit(X_train_vec, y_train)
print(nb_clf.score(X_test_vec,y_test))
```

```
0.5952197872613098
```

```
# Train a SVM classifier
svm_clf = LinearSVC(C = 1, max_iter = 100000)
svm_clf.fit(X_train_vec, y_train)
print(svm_clf.score(X_test_vec, y_test))
```

```
0.6284441881327695
```

```
# Print the confusion matrix for mnb and svm
print("Confusion Matrix for MNB Model", "\n")
print(confusion_matrix(y_test, nb_clf.predict(X_test_vec), labels = [0,1,2,3,4]), "\n")
print("Confusion Matrix for SVM Model", "\n")
print(confusion_matrix(y_test, svm_clf.predict(X_test_vec), labels = [0,1,2,3,4]), "\n")
```

```
Confusion Matrix for MNB Model

[[  809  1268   644    89    15]
 [  814  4404  5038   610    59]
 [  387  3034 24254  3574   348]
 [   55   520  5180  6489  1075]
 [    7    60   592  1899  1200]]
```

```
Confusion Matrix for SVM Model

[[  979  1305   464    61    16]
 [  857  4535  5012   473    48]
 [  239  2502 25948  2742   166]
 [   46   347  5397  6281  1248]
 [   10    29   444  1788  1487]]
```

```
Difference of SVM and MNB

[[ -170   -37   180    28    -1]
 [  -43  -131    26   137    11]
 [  148   532 -1694   832   182]
 [    9   173  -217   208  -173]
 [   -3    31   148   111  -287]]
```

```
# Print the precision, recall, and classification report for mnb
print("Precision for MNB Model")
print(precision_score(y_test, nb_clf.predict(X_test_vec), average = None), "\n")
print("Recall for MNB Model")
print(recall_score(y_test, nb_clf.predict(X_test_vec), average = None), "\n")
print("Classification Report for MNB Model")
print(classification_report(y_test, nb_clf.predict(X_test_vec), target_names = ["0","1","2","3","4"]), "\n")
```

```
# Print the precision, recall, and classification report for svm
print("Precision for SVM Model")
```

```
print(precision_score(y_test, svm_clf.predict(X_test_vec), average = None), "\n")
print("Recall for SVM Model")
print(recall_score(y_test, svm_clf.predict(X_test_vec), average = None), "\n")
print("Classification Report for SVM Model")
print(classification_report(y_test, svm_clf.predict(X_test_vec), target_names = ["0","1","2","3","4"]), "\n")
```

```
Precision for MNB Model
[0.39044402 0.47426233 0.67923154 0.51251876 0.44493882]

Recall for MNB Model
[0.28637168 0.40311213 0.76760452 0.48719874 0.31931879]

Classification Report for MNB Model
              precision    recall  f1-score   support

           0       0.39      0.29      0.33      2825
           1       0.47      0.40      0.44     10925
           2       0.68      0.77      0.72     31597
           3       0.51      0.49      0.50     13319
           4       0.44      0.32      0.37      3758

    accuracy                           0.60     62424
   macro avg       0.50      0.45      0.47     62424
weighted avg       0.58      0.60      0.58     62424
```

```
Precision for SVM Model
[0.45940873 0.52018812 0.69631021 0.55363596 0.50151771]

Recall for SVM Model
[0.34654867 0.41510297 0.8212172  0.47158195 0.3956892 ]

Classification Report for SVM Model
              precision    recall  f1-score   support

           0       0.46      0.35      0.40      2825
           1       0.52      0.42      0.46     10925
           2       0.70      0.82      0.75     31597
           3       0.55      0.47      0.51     13319
           4       0.50      0.40      0.44      3758

    accuracy                           0.63     62424
   macro avg       0.55      0.49      0.51     62424
weighted avg       0.61      0.63      0.62     62424
```

```
# Save the feature ranks
feature_ranks_very_neg_mnb = sorted(zip(nb_clf.feature_log_prob_[0], unigram_count_vectorizer.get_feature_names()))
feature_ranks_very_pos_mnb = sorted(zip(nb_clf.feature_log_prob_[4], unigram_count_vectorizer.get_feature_names()))
feature_ranks_very_neg_svm = sorted(zip(svm_clf.coef_[0], unigram_count_vectorizer.get_feature_names()))
feature_ranks_very_pos_svm = sorted(zip(svm_clf.coef_[4], unigram_count_vectorizer.get_feature_names()))

# Top 10 very negative for mnb and svm
print("Top 10 Indicative Words for Very Negative Category MNB Model", "\n")
print(feature_ranks_very_neg_mnb[-10: ], "\n")
print("Top 10 Indicative Words for Very Negative Category SVM Model", "\n")
print(feature_ranks_very_neg_svm[-10: ])

# Top 10 very positive for mnb and svm
print("Top 10 Indicative Words for Very Positive Category MNB Model", "\n")
print(feature_ranks_very_pos_mnb[-10: ], "\n")
print("Top 10 Indicative Words for Very Positive Category SVM Model", "\n")
print(feature_ranks_very_pos_svm[-10: ])
```

Top 10 "very negative"

```
Top 10 Indicative Words for Very Negative Category MNB Model

[(-7.272981555830627, 'cloying'), (-7.107467117353054, 'anachronistic'), (-6.980311941867808,
'phrase'), (-6.854271220972443, 'retaliatory'), (-6.802977926584893, 'animals'),
(-6.685194890928509, 'purists'), (-6.674145054741924, 'innocuous'), (-6.550275573029138,
'generosity'), (-5.560987055071435, 'confession'), (-5.511681294087281, 'undermining')]

Top 10 Indicative Words for Very Negative Category SVM Model

[(1.5328964959377875, 'smash'), (1.5428904836258428, 'atypically'), (1.5638132371261853,
'proper'), (1.6406562256319144, 'predicament'), (1.653080554360697, 'results'),
(1.6983639388516554, 'meddles'), (1.7061592497568103, 'stereotyped'), (1.7293156274197172,
'relevance'), (1.7771190732328035, 'complications'), (1.8892389624406014, 'precocious')]
```

Top 10 "very positive"

```
Top 10 Indicative Words for Very Positive Category MNB Model

[(-6.957636910776074, 'floria'), (-6.957636910776074, 'tied'), (-6.9313196024587, 'reduce'),
(-6.820777728058877, 'pork'), (-6.8092169056578005, 'veil'), (-6.753336447263344, 'generosity'),
(-6.753336447263344, 'slasher'), (-6.264489730216129, 'innocuous'), (-5.798661472380873,
'defiance'), (-4.9347657205846325, 'undermining')]

Top 10 Indicative Words for Very Positive Category SVM Model

[(1.436821700569572, 'nuances'), (1.457456842397187, 'landau'), (1.46238167810755, 'operandi'),
(1.498729917876349, 'complicated'), (1.5015474212054305, 'nephew'), (1.5265681505548783,
'startled'), (1.5775369644247974, 'suffocation'), (1.6072914762140895, 'blacks'),
(1.7597574194063945, 'start'), (1.8745669194109558, 'drowsy')]
```

There was little to no difference in performance when adding bigram vectors. There were not any bigrams that appeared in the top 10 lists. The top 10 lists in this task were a lot different from the top 10 lists in the previous task.

Task 3 - Best Classifier

In this task, I will use grid search to find the best hyper parameters for the MNB and SVM models on the same 60/40 split that was used in the previous tasks, I will conduct a 10 fold cross validation using the best hyper parameters, and finally I will re-train the models on 100% of the data and make predictions on the Kaggle test set, submit to the Kaggle competition and report my rank on the leaderboards.

```
# Set up the vectorization pipeline
tfidf_pip = Pipeline([
    ("tfidf_vectorizer", TfidfVectorizer())
])
```

```
# Set up the naive bayes pipeline
mnb_pipeline = Pipeline([
    ("tfidf_pip", tfidf_pip),
    ("mnb", MultinomialNB())
])
```

```
# Set up the grid search params
grid_params = {
  "mnb__alpha": numpy.linspace(0.5, 1.5, 3),
  'mnb__fit_prior': [True, False],
  "tfidf_pip__tfidf_vectorizer__use_idf": [True, False],
```

```
  "tfidf_pip__tfidf_vectorizer__norm": [None, "l1", "l2"],
}
```

```
# Train model
nb_clf = GridSearchCV(mnb_pipeline, grid_params)
nb_clf.fit(X_train, y_train)
predictions = nb_clf.predict(X_test)
print("Train Score = ", nb_clf.best_score_)
print("Test Score = ", accuracy_score(y_test, predictions))
print("Best Params = ", nb_clf.best_params_)
```

```
Train Score =  0.6019479717418196
Test Score =  0.6081475073689606
Best Params =  {'mnb__alpha': 1.5, 'mnb__fit_prior': True, 'tfidf_pip__tfidf_vectorizer__norm':
None, 'tfidf_pip__tfidf_vectorizer__use_idf': False}
```

```
# Fit vectorizer and MNB with the params

vectorizer = TfidfVectorizer(
    norm = None,
    use_idf = False
)

nb_clf = MultinomialNB(alpha = 1.5, fit_prior = True)
```

```
# Initiate the split object
split = KFold(n_splits = 10, shuffle = True, random_state = 42)

# Initiate a counter
run_counter = 0

# Iterate through each fold and conduct the process
for train_index, test_index in split.split(data_train, data_train["Sentiment"]):

    # Seperate out the train and test sets
    train_data = data_train.iloc[train_index, :]
    test_data = data_train.iloc[test_index, :]

    # Separate out the predictors
    X_train = train_data["Phrase"]
    X_train_vec = vectorizer.fit_transform(X_train)
    X_test = test_data["Phrase"]
    X_test_vec = vectorizer.transform(X_test)

    # Seperate out the class labels
    y_train = train_data["Sentiment"].copy()
    y_test = test_data["Sentiment"].copy()

    # Train model and and make predictions
    nb_clf.fit(X_train_vec, y_train)
    predictions = nb_clf.predict(X_test_vec)

    # Store the information from the run
    test_score = accuracy_score(y_test, predictions)

    # Update the counter after each iteration
    run_counter += 1
```

```
    # Print out the results
    print("Run ", run_counter, " Completed")
    print("Test Score ", test_score)
```

```
Run  1  Completed
Test Score  0.6148276303985647
Run  2  Completed
Test Score  0.6138023836985774
Run  3  Completed
Test Score  0.6190567730360118
Run  4  Completed
Test Score  0.6099577085736255
Run  5  Completed
Test Score  0.6120082019736
Run  6  Completed
Test Score  0.6130334486735871
Run  7  Completed
Test Score  0.6090606177111367
Run  8  Completed
Test Score  0.6146994745610662
Run  9  Completed
Test Score  0.6084198385236448
Run  10  Completed
Test Score  0.6186723055235166
```

```
# Set up the vectorization pipeline
tfidf_pip = Pipeline([
    ("tfidf_vectorizer", TfidfVectorizer())
])
```

```
# Set up the naive bayes pipeline
svm_pipeline = Pipeline([
    ("tfidf_pip", tfidf_pip),
    ("svm", LinearSVC(max_iter = 10000))
])
```

```
# Set up the grid search params
grid_params = {
  "svm__C": numpy.linspace(1, 10, 3),
  "tfidf_pip__tfidf_vectorizer__use_idf": [True, False],
  "tfidf_pip__tfidf_vectorizer__norm": [None, "l1", "l2"],
}
```

```
# Train model
svm_clf = GridSearchCV(svm_pipeline, grid_params)
svm_clf.fit(X_train, y_train)
predictions = svm_clf.predict(X_test)
print("Train Score = ", svm_clf.best_score_)
print("Test Score = ", accuracy_score(y_test, predictions))
print("Best Params = ", svm_clf.best_params_)
```

```
Train Score =  0.629672306900651
Test Score =  0.6347718826092529
Best Params =  {'svm__C': 1.0, 'tfidf_pip__tfidf_vectorizer__use_idf': False}
```

```
# Fit vectorizer and SVM with the params

vectorizer = TfidfVectorizer()

svm_clf = LinearSVC()
```

```
# Initiate the split object
split = KFold(n_splits = 10, shuffle = True, random_state = 42)

# Initiate a counter
run_counter = 0

# Iterate through each fold and conduct the process
for train_index, test_index in split.split(data_train, data_train["Sentiment"]):

    # Seperate out the train and test sets
    train_data = data_train.iloc[train_index, :]
    test_data = data_train.iloc[test_index, :]

    # Separate out the predictors
    X_train = train_data["Phrase"]
    X_train_vec = vectorizer.fit_transform(X_train)
    X_test = test_data["Phrase"]
    X_test_vec = vectorizer.transform(X_test)

    # Seperate out the class labels
    y_train = train_data["Sentiment"].copy()
    y_test = test_data["Sentiment"].copy()

    # Train model and and make predictions
    nb_clf.fit(X_train_vec, y_train)
    predictions = nb_clf.predict(X_test_vec)

    # Store the information from the run
    test_score = accuracy_score(y_test, predictions)

    # Update the counter after each iteration
    run_counter += 1

    # Print out the results
    print("Run ", run_counter, " Completed")
    print("Test Score ", test_score)
```

```
Run  1  Completed
Test Score  0.6441112392669486
Run  2  Completed
Test Score  0.6433423042419583
Run  3  Completed
Test Score  0.6496219402793797
Run  4  Completed
Test Score  0.6423811354607203
Run  5  Completed
Test Score  0.6475714468794054
Run  6  Completed
Test Score  0.6477636806356529
Run  7  Completed
Test Score  0.6399461745482506
Run  8  Completed
Test Score  0.6455850313981802
Run  9  Completed
Test Score  0.6425092912982187
Run  10  Completed
Test Score  0.6496860181981289
```

```
# Kaggle submission
vectorizer = TfidfVectorizer(use_idf = False)
X_vec = vectorizer.fit_transform(X)
svm_clf = LinearSVC()
svm_clf.fit(X_vec, y)
X_kaggle = data_test.Phrase.copy()
ids = data_test.PhraseId.copy()
X_kaggle_vec = vectorizer.transform(X_kaggle)
predictions = svm_clf.predict(X_kaggle_vec)
submission = pandas.DataFrame(list(zip(
    ids, predictions)), columns = ["PhraseId", "Sentiment"])
submission.to_csv("submission.csv", index = False)
```

Task 3 - Conclusion

There were no major improvements that I found from tuning the hyperparameters. One of the challenges was that it was taking a long time for the grid search to complete, so I had to narrow down the options. I decided to use the SVM model for the Kaggle submission because it had a better accuracy score. When I submitted to Kaggle, the accuracy went down slightly. I am not surprised, because there was most likely some over fitting occuring with the SVM model. I think more work could be done in preprocessing the data and tuning the hyper parameters, but it would time to go through this process.