ML 135 - Project 02

Import libraries

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import string

from sklearn.feature_extraction import text
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPClassifier

from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

from collections import OrderedDict
```

Read data and preprocess

In this step, we input training and testing data and remove punctuation from all inputs using the translator class from Python's built in string library for better data uniformity.

In [2]:

```
x_train_df = pd.read_csv('data_reviews/x_train.csv')
y_train_df = pd.read_csv('data_reviews/y_train.csv')
x_test_df = pd.read_csv('data_reviews/x_test.csv')

x_train = x_train_df['text'].values.tolist()
y_train = y_train_df['is_positive_sentiment'].values.tolist()
x_test = x_test_df['text'].values.tolist()

# remove punctuation
translator = str.maketrans('','',string.punctuation)
x_train = [i.translate(translator) for i in x_train]
x_test = [i.translate(translator) for i in x_test]

#example input
print('Example Input: ' + x_train[0])
```

```
Example Input: Oh and I forgot to also mention the weird color effec
t it has on your phone
```

Part One: Classifying Review Sentiment with Bag-of-Words Features

1.1 Generating BoW features

To extract features from the input reviews we will convert the text to a matrix of features using the bag of words representation. First step is to preprocess the input data using the following pipeline: Removing punctuation from all inputs (done in previous step), Converting all input to lowercase, removing accents from characters and removing stopwords using the nltk english stopword list. These preprocessing steps are useful for better uniformity in our input data. Stopwords will be handled because the classifiers might benefit from not tokenizing words in an example which are uninformative regarding to the sentiment of that example such as ('the', 'he', 'she'), in order to reduce the number of features necessary to represent the semantics of a review. To generate a BoW input type, the CountVectorizer class will be used from sklearn.feature_extraction. CountVectorizer will convert a collection of text documents (input) to a matrix of token counts, which is a weight that represents how ofter a token (word) is seen in the whole input dataset. For a first inspection, I chose not to use the TfidfVectorizer, which consideres the inverse document frequency weight, that negatively weighs features that appear often in documents. It might be useful to examine logistic regression with inverse frequency, but for now the baseline word frequency vectorizer will be considered. As you can see the BoW representations for each training and testing example are arrays of 4564 features. Having this amount of features might restrict classifier performance in terms of timing, but because the training dataset is not that large, I don't think it will be a problem.

In [24]:

```
from nltk.corpus import stopwords

# Feature generation pipeline for Bag of Words representation
def convert_to_BoW(x_train, y_train, x_test = None):
    s_w = set(stopwords.words('english'))

    count_vec = text.CountVectorizer(lowercase=True, strip_accents='ascii', ngra
m_range=(1,1),
                                     stop_words=s_w, analyzer='word')
    count_vec.fit(x_train)

    # Convert training and testing data to BoW feature form
    x_train_bow = count_vec.transform(x_train.copy())
    y_train_bow = np.array(y_train)

    if x_test is not None:
        x_test_bow = count_vec.transform(x_test.copy())
    else:
        x_test_bow = None

    return x_train_bow, y_train_bow, x_test_bow

x_train_bow, y_train_bow, x_test_bow = convert_to_BoW(x_train.copy(), y_train.co
py(), x_test.copy())
print('Shape of training dataset: ' + str(x_train_bow.shape))
print('Shape of testing dataset: ' + str(x_test_bow.shape))
```

```
Shape of training dataset: (2400, 4564)
Shape of testing dataset: (600, 4564)
```

1.2 Generating Logistic Regression model

Let's start by generating a baseline logistic regression model using LogisticRegression from sklearn.linear_model. Default value for tolerance for stopping criteria (tol = 1e-4) is selected, with the liblinear solver (solver=liblinear), for a default maximum number of iterations (max_iter = 100). The model is fit on the BoW representation inputs that were created in the previous step. I will vary hyperparameter C, which specifies the inverse of regularization strength (1/lambda). Small values of C correspond to stronger data regularization so by varying C I will be able to evaluate the effect of C on model overfitting. We will try 10-fold cross validation (cv=10) along with the grid search for best value of C. This will be done using sklearn's built-in GridSearchCV. The values that will be considered for C will be the following: [0.0001, 0.001, 0.01, 0.1, 1, 10]. I decided to vary C by factors of 10, so as to incrementaly inspect larger magnitude values with an appropriate step size. The results will be shown in a dataframe as well as graphically. I inspect the scores for each cross validation split for both the training sets and the validation sets (inputs that are kept out from training at each split).

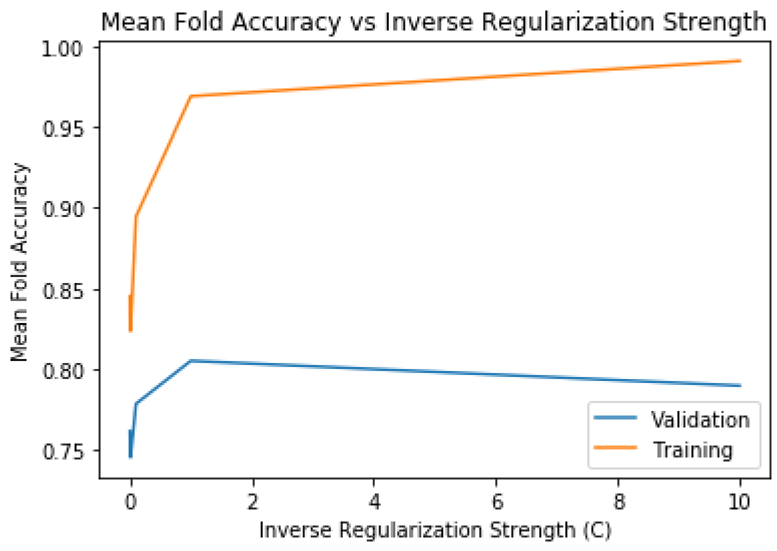
In [5]:

```
model_reg = LogisticRegression(tol=1e-4, C=1.0, solver='liblinear', max_iter=100
)
c_grid = {'C':[0.0001, 0.001, 0.01, 0.1, 1, 10]}
results = GridSearchCV(model_reg, c_grid, refit=True, cv=10, return_train_score=
True)
results.fit(x_train_bow, y_train_bow)
df = pd.DataFrame(results.cv_results_)

test_scores = results.cv_results_['mean_test_score']
train_scores = results.cv_results_['mean_train_score']

plt.plot(c_grid['C'], test_scores)
plt.plot(c_grid['C'], train_scores)
plt.title('Mean Fold Accuracy vs Inverse Regularization Strength')
plt.xlabel('Inverse Regularization Strength (C)')
plt.ylabel('Mean Fold Accuracy')
plt.legend(('Validation', 'Training'))

is_positive_reg = results.predict_proba(x_test_bow)[:, 1] # Predicted proba valu
es for gradescope
np.savetxt('results_reg.txt', is_positive_reg)
```



```
In [6]:
```

```
tmp = pd.DataFrame(data={'C': c_grid['C'],'Training': df['std_train_score'], 'Te
sting': df['std_test_score'],
                        'Mean Train Accuracy': df['mean_train_score'], 'Mean Tes
t Accuracy': df['mean_test_score']})
print('Standard deviation')
print(tmp)
print("Best cross-validation score: {:.2f}".format(results.best_score_))
print('Best Hyperparameter Index: ' + str(results.best_index_))
print('Best LR Model: ' + str(results.best_estimator_))
```

```
Standard deviation
         C  Training   Testing  Mean Train Accuracy  Mean Test Accur
acy
0   0.0001  0.006359  0.027563             0.845139            0.761
667
1   0.0010  0.003395  0.037722             0.833426            0.759
167
2   0.0100  0.005471  0.043159             0.823426            0.745
417
3   0.1000  0.002739  0.029333             0.894583            0.778
333
4   1.0000  0.001958  0.032156             0.968935            0.805
000
5  10.0000  0.001284  0.031083             0.990787            0.789
583
Best cross-validation score: 0.81
Best Hyperparameter Index: 4
Best LR Model: LogisticRegression(C=1, class_weight=None, dual=Fals
e, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.000
1, verbose=0,
                   warm_start=False)
```

The graph shows the variation of the mean accuracy with respect to C, for both the training fold/split and the validation data (rest). The table shows the average standard deviation of the accuracy scores for each C for both training and testing folds. As you can see, accuracy for the training fold is consistently higher than that for validation data because the curve for the training data is always higher than the curve for the validation data in the y-axis. Initially, there is a small cusp with a worst accuracy at C = 0.01. The mean accuracy curve of the validation data is greatest at C = 1, and then decreases as C increases because training data regularization decreases and overfitting occurs. That is why the training accuracy keeps increasing but the validation accuracy does not. The model does not generalize well for high value of C's for the validation data, so such large values for C should be avoided in the current tuning. With this hyperparameter grid search and through cross validation, we were able to achieve a best mean validation accuracy of 81%.

1.3 Generating Neural Network

I will now define a neural network for the BoW features using scikit's MLPClassifier. This model will be explored using the same 10-fold cross-validation technique as the logistic regression model previously. GridSearchCV will be used to evaluate the effect of L2 Regularization penalty (alpha) on the model's accuracy on validation data (mean accuracy across all cross validation folds). I assume that an increase of the L2 regularization penalty will lead to a decrease in overfitting and a better generalization for the validationn set. The values for alpha that will be inspected are: [0.0001, 0.001, 0.01, 0.1, 1]. I selected these values with a similar reasoning to the previous Logistic Regression hyperparameter search. With a cv parameter of 10 for GridSearch, the model is going to fit 10 folds for our 5 parameters for a total of 50 fits.

In [7]:

```
model_mlp = MLPClassifier(max_iter= 50, learning_rate='adaptive')
param_grid = {'alpha': [0.0001, 0.001, 0.01, 0.1, 1], 'hidden_layer_sizes': [(10
0, )]}
results_mlp = GridSearchCV(model_mlp, param_grid, refit=True, cv=10, return_trai
n_score=True, n_jobs= -1, verbose=2)
results_mlp.fit(x_train_bow, y_train_bow)
df = pd.DataFrame(results_mlp.cv_results_)
```

```
Fitting 10 folds for each of 5 candidates, totalling 50 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent w
orkers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:  2.1min
[Parallel(n_jobs=-1)]: Done  50 out of  50 | elapsed:  3.2min finish
ed
/Users/chrismits/miniconda3/envs/ml135_env/lib/python3.7/site-packag
es/sklearn/neural_network/multilayer_perceptron.py:566: ConvergenceW
arning: Stochastic Optimizer: Maximum iterations (50) reached and th
e optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

In [8]:

```
print('Best Classifier Settings: '),
print(results_mlp.best_estimator_) # best MLPCLassifier from gridsearch
print('Best index: '),
print(results_mlp.best_index_) # best param combo
print("Best cross-validation score: {:.2f}".format(results_mlp.best_score_))
print('\nResults:\n')
tmp = pd.DataFrame(data={'alpha': df['param_alpha'], 'Std Score Training': df['s
td_train_score'],
                         'Std Score Testing': df['std_test_score'], 'Mean Train
 Accuracy': df['mean_train_score'],
                         'Mean Test Accuracy': df['mean_test_score']})
print(tmp)
```

```
Best Classifier Settings:
MLPClassifier(activation='relu', alpha=1, batch_size='auto', beta_1=
0.9,
              beta_2=0.999, early_stopping=False, epsilon=1e-08,
              hidden_layer_sizes=(100,), learning_rate='adaptive',
              learning_rate_init=0.001, max_iter=50, momentum=0.9,
              n_iter_no_change=10, nesterovs_momentum=True, power_t=
0.5,
              random_state=None, shuffle=True, solver='adam', tol=0.
0001,
              validation_fraction=0.1, verbose=False, warm_start=Fal
se)
Best index:
4
Best cross-validation score: 0.80

Results:

    alpha  Std Score Training  Std Score Testing  Mean Train Accurac
y  \
0  0.0001            0.000780           0.026709             0.99509
3
1   0.001            0.000593           0.030279             0.99518
5
2    0.01            0.000578           0.029709             0.99481
5
3     0.1            0.001149           0.029179             0.99361
1
4       1            0.001808           0.033657             0.97939
8

   Mean Test Accuracy
0            0.787083
1            0.786250
2            0.789167
3            0.789167
4            0.802083
```

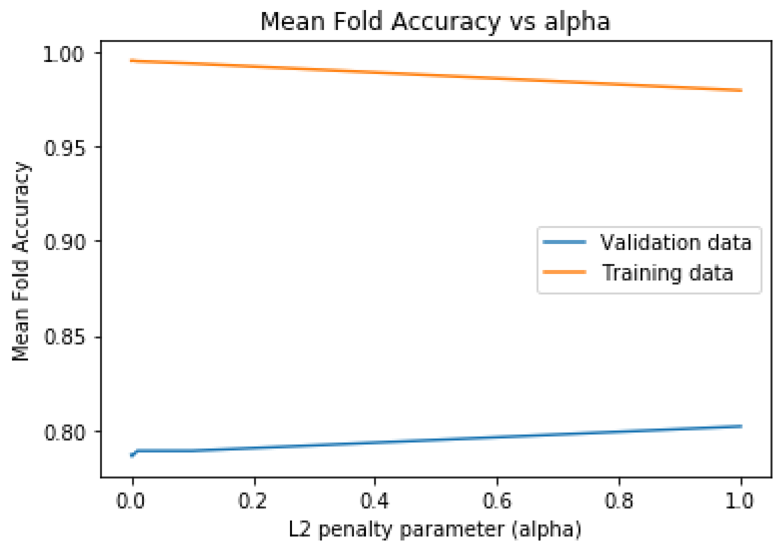
The hyperparameter combination with the highest mean cross-validated score, is that of index 4 as seen above, with an alpha of 1 for an accuracy of 80%. Observe that the standard deviation of the score for each combination in the training set increases with alpha. Let's plot accuracy with respect to alpha:

In [9]:

```
test_scores_mlp = results_mlp.cv_results_['mean_test_score']
train_scores_mlp = results_mlp.cv_results_['mean_train_score']

plt.plot(df['param_alpha'], test_scores_mlp)
plt.plot(df['param_alpha'], train_scores_mlp)
plt.title('Mean Fold Accuracy vs alpha')
plt.xlabel('L2 penalty parameter (alpha)')
plt.ylabel('Mean Fold Accuracy')
plt.legend(('Validation data', 'Training data'))

is_positive_mlp = results_mlp.predict_proba(x_test_bow)[:, 1] # Predicted proba
 values for gradescope
np.savetxt('results_mlp.txt', is_positive_mlp)
```



As alpha increases, the mean fold accuracy for the training set decreases as regularization weight of data increases. Conversely, the mean fold accuracy for the validation set increases as alpha increases because the model generalizes better for the new data with higher alpha's.

1.4 Generating Third Model (SVM classifier)

For this step I will use Scikit's built in SVC (C Support vector classification), found in scikit's SVM package, with BoW inputs. We will use GridSearchCV, to explore the hyperparameter space as well as to perform StratifiedKFold cross validation. The SVC model will use a linear kernel, a balanced class_weight parameter. To predict the probabilities of the testing dataset, probability needs to be set to true in SVC for probability estimates to be enabled. I will then vary the hyperparameter C which is the penalty parameter of the error term. Let's attempt to fit the model for the following values of C : [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]. For our total of 5 cross validation splits, the model will be fit 40 times.

In [10]:

```
param_grid_svm = {'C': [0.01, 0.1, 1, 10, 100, 1000, 10000]}
results_svm = GridSearchCV(SVC(probability=True), param_grid_svm, cv=5, verbose=
1, n_jobs=-1, refit=True, return_train_score=True)
results_svm.fit(x_train_bow, y_train_bow)
df = pd.DataFrame(results_svm.cv_results_)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent w
orkers.

Fitting 5 folds for each of 7 candidates, totalling 35 fits

[Parallel(n_jobs=-1)]: Done  35 out of  35 | elapsed:    9.6s finish
ed
/Users/chrismits/miniconda3/envs/ml135_env/lib/python3.7/site-packag
es/sklearn/svm/base.py:193: FutureWarning: The default value of gamm
a will change from 'auto' to 'scale' in version 0.22 to account bett
er for unscaled features. Set gamma explicitly to 'auto' or 'scale'
to avoid this warning.
  "avoid this warning.", FutureWarning)
```

In [11]:

```
print('Best Classifier Settings: '),
print(results_svm.best_estimator_) # best MLPCLassifier from gridsearch
print('Best index: '),
print(results_svm.best_index_) # best param combo
print("Best cross-validation score: {:.2f}".format(results_svm.best_score_))
print('Analytical results for Standard deviation and mean fold accuracy')
tmp = pd.DataFrame(data={'C': df['param_C'], 'Std Train Dev': df['std_train_scor
e'], 'Std Test Dev': df['std_test_score'],
                              'Mean Train Accuracy': df['mean_train_score'], 'Mean Te
st Accuracy': df['mean_test_score']})
print(tmp)
```

```
Best Classifier Settings:
SVC(C=1000, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecate
d',
    kernel='rbf', max_iter=-1, probability=True, random_state=None,
    shrinking=True, tol=0.001, verbose=False)
Best index:
5
Best cross-validation score: 0.80
Analytical results for Standard deviation and mean fold accuracy
       C  Std Train Dev  Std Test Dev  Mean Train Accuracy  Mean Tes
t Accuracy
0   0.01       0.094762      0.088819             0.637604
0.620833
1    0.1       0.094762      0.088819             0.637604
0.620833
2      1       0.094762      0.088819             0.637604
0.620833
3     10       0.051316      0.052793             0.643021
0.612917
4    100       0.017274      0.043549             0.841146
0.735833
5   1000       0.003350      0.025055             0.972813
0.795000
6  10000       0.001010      0.031988             0.992604
0.765000
```

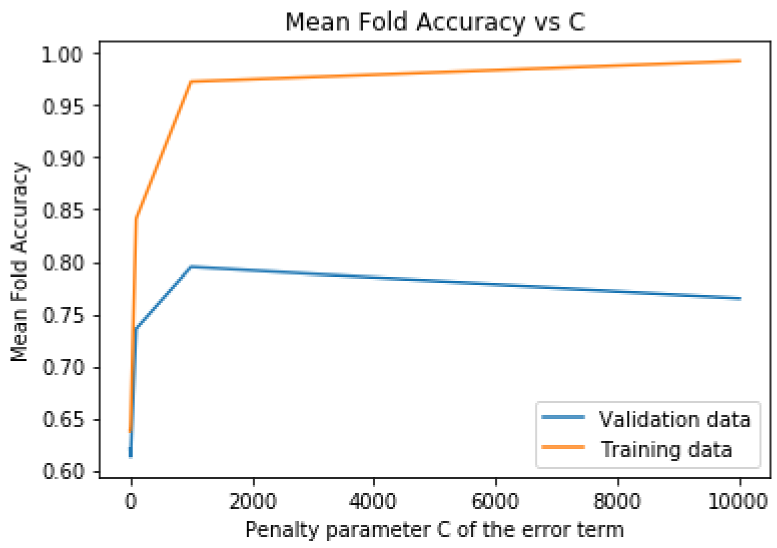
As we can see from the analytical results above and the accuracy graph below, it seems like as C increases the accuracy for both training and testing sets also increase, up to a point after which we see a decrease in mean test accuracy. What is also apparent is that for C values less than 1 that were inspected, accuracy and standard error remained the same, which means that penalty parameter for values less than 1 does not affect the results. For low values of penalty parameter C, accuracy is low. There is a large jumo from C = 10 to c = 100 which leads to increase in accuracy and decrease in the variation between the values. The general trend is understood in the graph below. As C increases from that point onwards, the weight is too high which burdens generalization to testing data. This is why we see the gradual increasing slope in training data and decreasing slope for validation data.

In [12]:

```
test_scores_svm = results_svm.cv_results_['mean_test_score']
train_scores_svm = results_svm.cv_results_['mean_train_score']

plt.plot(df['param_C'], test_scores_svm)
plt.plot(df['param_C'], train_scores_svm)
plt.title('Mean Fold Accuracy vs C')
plt.xlabel('Penalty parameter C of the error term')
plt.ylabel('Mean Fold Accuracy')
plt.legend(('Validation data', 'Training data'))

is_positive_svm = results_svm.predict_proba(x_test_bow)[:, 1] # Predicted proba
 values for gradescope
np.savetxt('results_svm.txt', is_positive_svm)
```



1.5 Best Classifier

All three of the models reached a best cross validation score of about 80%. Starting with Logistic Regression, it had the best mean cross validation score for the testing data (dataset that was left out at each fit) for the inverse regularization constant C = 1, with an accuracy of 80.5%. We saw that it was prone to overfitting for larger C, as training accuracy increased at a much higher rate than testing accuracy as C increased. The neural network model was much more rigid with respect to changes in hyperparameter alpha, with the training set accuracy being consistently high. As alpha increased however, training accuracy decreased and testing accuracy increased, a good sign for the reduction of overfitting and the increase in generalization. However, this MLP model was much harder to evaluate hyperparameters because it is less flexible time-wise. A single fit requires much more time, plus it is harder and more time-consuming to consistently vary hyperparameters and inspect results, due to the increased requirement in computational resources. The SVM classifier was faster than MLP model to run and about the same time as the logistic regression model, but with slightly worse accuracy results for the testing data. Note that signs of overfitting where not apparent for the neural network which goes to show that with a little extra further inspection and maybe a grid search on hyperparameters such as alpha, hidden layer size, adaptive learning rate and tolerance better results might be expected, at the expense of time and computational resources. Since everything is a tradeoff however, the best model for cross validation data was the logistic regression model with C = 1. Let's see if this results generalizes in the testing data.

In [13]:

```
def calc_confusion_matrix_for_threshold(y_true, y_proba, thresh):
    cm = confusion_matrix(y_true, y_proba >= thresh)
    cm_df = pd.DataFrame(data=cm, columns=[0, 1], index=[0, 1])
    cm_df.columns.name = 'Predicted'
    cm_df.index.name = 'True'
    return cm_df

best_model_lr = LogisticRegression(C=1, class_weight=None, dual=False, fit_inter
cept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbose=0,
                   warm_start=False)

x_train_lr, x_test_lr, y_train_lr, y_test_lr = train_test_split(x_train, y_train
,
                                                                 test_size=0.2, r
andom_state=4)

x_train_lr_bow, y_train_lr_bow, x_test_lr_bow = convert_to_BoW(x_train_lr, y_tra
in_lr, x_test_lr)

best_model_lr.fit(x_train_lr_bow, y_train_lr_bow)
y_pred_lr = best_model_lr.predict_proba(x_test_lr_bow)[:,1]

misclassified = np.where(y_test_lr != best_model_lr.predict(x_test_lr_bow))

print('Confusion Matrix for Logistic Regression:')
print(calc_confusion_matrix_for_threshold(y_test_lr, y_pred_lr, 0.5))

tmp = []
fp = 0
fn = 0
for i in misclassified[0]:
    if y_test_lr[i] == 1:
        tmp.append(0)
    else:
        tmp.append(1)

res_df = pd.DataFrame(data={'Review': [x_test_lr[i] for i in misclassified[0]],
                        'Output': tmp})
print(res_df)
```

```
Confusion Matrix for Logistic Regression:
Predicted    0    1
True
0          198   36
1           56  190
                                               Review  Output
0                                         Go rent it        0
1                  All in all a great disappointment        1
2                    So far it has worked like a charm       0
3   If you check the directors filmography on this...       1
4               The selection of food was not the best       1
..                                                 ...     ...
87               Someone shouldve invented this sooner       0
88                               I cant wait to go back       0
89  It was a riot to see Hugo Weaving play a sexob...       0
90                          It has been a winner for us       0
91                    Ill be looking for a new earpiece       1

[92 rows x 2 columns]
```

The table above contains some examples that were misclassified by the Logistic Regression model. The distribution of review sources is pretty much equal for imdb, amazon and yelp meaning it does not seem like the classifier performs poorly for some features consistently. As seen in the confusion matrix above, the classifier has a tendency to misclassify positive sentiment, judging by the high amount of false negatives. ADD SOME MORE EXPLANATION AS TO WHY THESE RESULTS?

1.6 Test best classifier on leaderboard

When the results were tested against the leaderboard, an error rate of 0.202 was reported with an area under ROC curve of 0.871. The error rate matches the performance seen in cross-validation, but does not match the results on training data as training data error rate was much lower. This makes it clear that the model overfits. What the AUROC tells us is that there is an 87% chance that the model will distinguish between a positive and a negative review, which is not ideal (1) but is respectable.

Part Two: Classifying Review Sentiment with Word Embeddings

Feature generation pipeline

Our token generation pipeline has similar preprocessing steps as in the first part. First of all, case is unified by converting all characters to lowercase. Stopwords are ignored using the nltk stopwords list. After this, input strings are tokenized, that is split by whitespace and converted to token lists. All tokens with length less than or equal to one are ignored. Finally, each token (word) is converted to an array of weights, following the word2vec dictionary mapping, if there is one. After the cleaning step, we are left with the training and testing dataset where each review now consists of a vector of weights. To go from a word embedding to a sentence embedding we will try out a simple conversion first: Each vector will be averaged, so that a review is now a weighted vector containing the average of the weights of each word in review. Now, when a word appears multiple times, it is weighted more strongly. Through our stopwords we ignore common words, but not much is done to ignore rare words or think about the document frequency of the words in our datasets. The feature generation is done on training and testing set and we are left with our future model inputs, x_train_we and x_test_we.

In [14]:

```
def clean_convert_to_we(x_input, word2vec):
    x_input = [i.lower() for i in x_input]
    #get stopwords
    s_w = set(stopwords.words('english'))
    tmp = []

    placeholder = [0 for i in range(50)]

    for i in x_train:
        tokens = i.split()
        tokens = [j for j in tokens if j not in s_w]
        tokens = [j for j in tokens if len(j) > 1]

        tokens = [word2vec[j] if j in word2vec else placeholder for j in tokens]
        tmp.append(tokens)

    return tmp

# Import Word Embeddings
word_embeddings = pd.read_csv('pretrained_word_embeddings/glove.6B.50d.txt.zip',
header=None, sep=' ', index_col=0,
                              nrows=100000, compression='zip', encoding='utf-8',
quoting=3)
word_list = word_embeddings.index.values.tolist()
word2vec = OrderedDict(zip(word_list, word_embeddings.values))

x_train_we = clean_convert_to_we(x_train, word2vec)
x_test_we = clean_convert_to_we(x_test, word2vec)

# average all word2vec embeddings
for i in range(len(x_train_we)):
    x_train_we[i] = [np.average(col) for col in zip(*x_train_we[i])]
    x_test_we[i] = [np.average(col) for col in zip(*x_test_we[i])]

x_train_we = np.array(x_train_we)
x_test_we = np.array(x_test_we)
y_train_we = np.array(y_train.copy())

print('Shape of Training data: ' + str(x_train_we.shape))
print('Shape of Testing data: ' + str(x_test_we.shape))
print('Shape of Training data Y: ' + str(y_train_we.shape))
```

```
Shape of Training data: (2400, 50)
Shape of Testing data: (2400, 50)
Shape of Training data Y: (2400,)
```

After running the feature generation pipeline for the word embeddings model, we end up with inputs with 50 features each. Now, we can go ahead and evaluate performance of these inputs on different models, starting with a Logistic Regression classifier.

2.2 Generating Logistic Regression model

I will now define a baseline LR model. For our constant parameters, I will use the liblinear solver, with a stopping tolerance of 1e-4 for a maximum iteration value of 100.

In [15]:

```
model_reg_we = LogisticRegression(tol=1e-4, solver='liblinear', max_iter=100)
```

As in the Logistic Regression model using the Bag of Words feature representation, I will inspect the effect of the inverse of regularization strength C, on the performance of the classifier for the word embeddings representation. The values of C inspected will be [0.0001, 0.001, 0.01, 0.1, 1, 10]. The performance of the classifier will be measured through cross validation for all values of C, using sklearn's GridSearchCV for a cross validation split of 10. The range of values explored in C are enough to evaluate overfitting in the model and are the same values inspected in 1.2, for comparisons to be allowed between the two feature representation techniques.

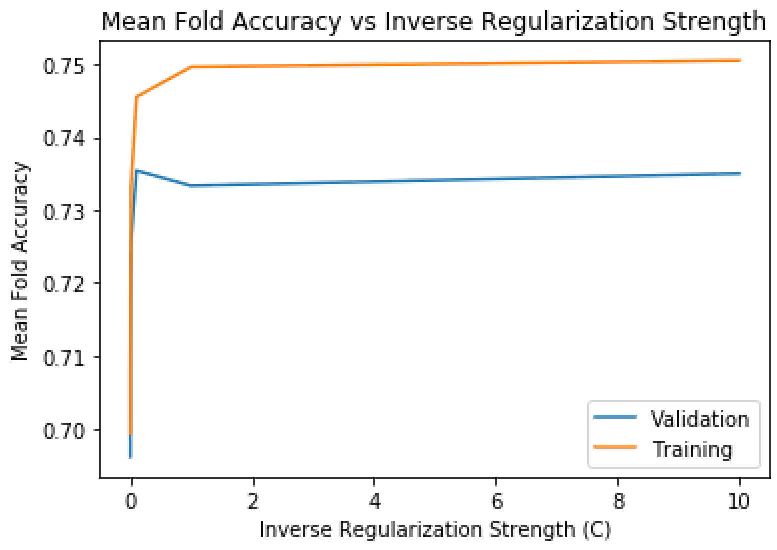
In [16]:

```
c_grid = {'C':[0.0001, 0.001, 0.01, 0.1, 1, 10]}
results_reg_we = GridSearchCV(model_reg_we, c_grid, refit=True, cv=10, return_tr
ain_score=True)
results_reg_we.fit(x_train_we, y_train_we)
df = pd.DataFrame(results_reg_we.cv_results_)

test_scores = results_reg_we.cv_results_['mean_test_score']
train_scores = results_reg_we.cv_results_['mean_train_score']

plt.plot(c_grid['C'], test_scores)
plt.plot(c_grid['C'], train_scores)
plt.title('Mean Fold Accuracy vs Inverse Regularization Strength')
plt.xlabel('Inverse Regularization Strength (C)')
plt.ylabel('Mean Fold Accuracy')
plt.legend(('Validation', 'Training'))

is_positive_reg_we = results_reg_we.predict_proba(x_test_we)[:, 1] # Predicted p
roba values for gradescope
np.savetxt('results_reg_we.txt', is_positive_reg_we)
```



```
In [17]:
```

```
tmp = pd.DataFrame(data={'C': c_grid['C'],'Training': df['std_train_score'], 'Te
sting': df['std_test_score'],
                        'Mean Train Accuracy': df['mean_train_score'], 'Mean Tes
t Accuracy': df['mean_test_score']})
print('Standard deviation')
print(tmp)
print("Best cross-validation score: {:.2f}".format(results_reg_we.best_score_))
print('Best Hyperparameter Index: ' + str(results_reg_we.best_index_))
print('Best LR Model: ' + str(results_reg_we.best_estimator_))
```

```
Standard deviation
         C  Training   Testing  Mean Train Accuracy  Mean Test Accur
acy
0   0.0001  0.010160  0.043837             0.699398            0.696
250
1   0.0010  0.005766  0.039704             0.715787            0.708
333
2   0.0100  0.005548  0.041602             0.733194            0.725
417
3   0.1000  0.005682  0.050561             0.745509            0.735
417
4   1.0000  0.004411  0.047398             0.749630            0.733
333
5  10.0000  0.004276  0.047037             0.750509            0.735
000
Best cross-validation score: 0.74
Best Hyperparameter Index: 3
Best LR Model: LogisticRegression(C=0.1, class_weight=None, dual=Fal
se, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.000
1, verbose=0,
                   warm_start=False)
```

As you can see in the detailed results above, the best cross-validation score was for C = 0.1, with an accuracy of 76% on the left out data. Note that this is lower than the accuracy achieved using the bag of words representation. Like the bag of words representation for LR model, the accuracy of the training dataset is consistently higher than that of testing dataset, which is expected. What is different is that the values of the training accuracy right now are consistently less than the values of the BoW representation. The training accuracy is also significantly close to testing accuracy, which is good in terms of generalization and the avoidance of overfitting. However, the error rate on the testing set is always higher. Therefore, we can make the characterization that so far, the word embedding representation is more resistant to overfitting and generalizes better for unseen data.

2.3 Generating Neural Network

For the word embedding feature representation, I will evaluate the effect of different combinations of momentum for gradient descent update and learning rate (step size in updating weights) on the classification accuracy of the model. To do this I first create an MLPClassifier with a constant learning rate and stochastic gradient descent for weight optimization (solver='sgd'). I will try a range of momentum values: [0.0, 0.2, 0.4, 0.6, 0.8, 0.9] with 0.9 being sklearn's default value, since momentum can be between 0 and 1.I chose two different learning rates: [0.001, 0.01] with 0.001 being sklearn's default. I won't inspect other values because for larger learning rate I had not found any success in my previous experiments. Since learning rate affects the amount of weight update and momentum tunes how much that weight update will affect the next, trying out different combinations of these values might lead to interesting results. I will split the training data into 10 folds and perform cross validation on every combination of momentum and learning rate using GridSearchCV with the cv parameter set to 10. Other than that, the parameters for GridSearchCV remain the same as in previous models, with the exception of the different model and parameter grid.

In [18]:

```
model_mlp_we = MLPClassifier(learning_rate='constant', solver='sgd', nesterovs_m
omentum=True)
param_grid = {'momentum': [0.0, 0.2, 0.4, 0.6, 0.8, 0.9], 'learning_rate_init':[
0.001, 0.01]}
results_mlp_we = GridSearchCV(model_mlp_we, param_grid, refit=True, cv=10, retur
n_train_score=True, n_jobs= -1, verbose=2)
results_mlp_we.fit(x_train_we, y_train_we)
df = pd.DataFrame(results_mlp_we.cv_results_)
```

```
Fitting 10 folds for each of 12 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent w
orkers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:   11.7s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:   45.0s finish
ed
/Users/chrismits/miniconda3/envs/ml135_env/lib/python3.7/site-packag
es/sklearn/neural_network/multilayer_perceptron.py:566: ConvergenceW
arning: Stochastic Optimizer: Maximum iterations (200) reached and t
he optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

In [19]:

```
print('Best Classifier Settings: '),
print(results_mlp_we.best_estimator_) # best MLPCLassifier from gridsearch
print('Best index: '),
print(results_mlp_we.best_index_) # best param combo
print("Best cross-validation score: {:.2f}".format(results_mlp_we.best_score_))
print('\nResults:\n')
tmp = pd.DataFrame(data={'momentum': df['param_momentum'], 'learning rate': df[
'param_learning_rate_init'], 'Std Score Training': df['std_train_score'],
                          'Std Score Testing': df['std_test_score'], 'Mean Train
 Accuracy': df['mean_train_score'],
                         'Mean Test Accuracy': df['mean_test_score']})
print(tmp)
```

```
Best Classifier Settings:
MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', be
ta_1=0.9,
              beta_2=0.999, early_stopping=False, epsilon=1e-08,
              hidden_layer_sizes=(100,), learning_rate='constant',
              learning_rate_init=0.01, max_iter=200, momentum=0.9,
              n_iter_no_change=10, nesterovs_momentum=True, power_t=
0.5,
              random_state=None, shuffle=True, solver='sgd', tol=0.0
001,
              validation_fraction=0.1, verbose=False, warm_start=Fal
se)
Best index:
11
Best cross-validation score: 0.74

Results:

   momentum learning rate  Std Score Training  Std Score Testing  \
0         0         0.001            0.023172           0.043748
1       0.2         0.001            0.024850           0.039045
2       0.4         0.001            0.015482           0.045308
3       0.6         0.001            0.019582           0.043062
4       0.8         0.001            0.006481           0.048126
5       0.9         0.001            0.006383           0.043740
6         0          0.01            0.006087           0.048464
7       0.2          0.01            0.005455           0.052356
8       0.4          0.01            0.004887           0.043916
9       0.6          0.01            0.005637           0.050249
10      0.8          0.01            0.005553           0.050888
11      0.9          0.01            0.004541           0.035843

    Mean Train Accuracy  Mean Test Accuracy
0              0.637176            0.631667
1              0.651528            0.634583
2              0.666944            0.660000
3              0.700046            0.684583
4              0.721667            0.713750
5              0.738380            0.720833
6              0.739167            0.724583
7              0.739630            0.728750
8              0.745185            0.729583
9              0.751157            0.735000
10             0.759491            0.734167
11             0.777778            0.741667
```

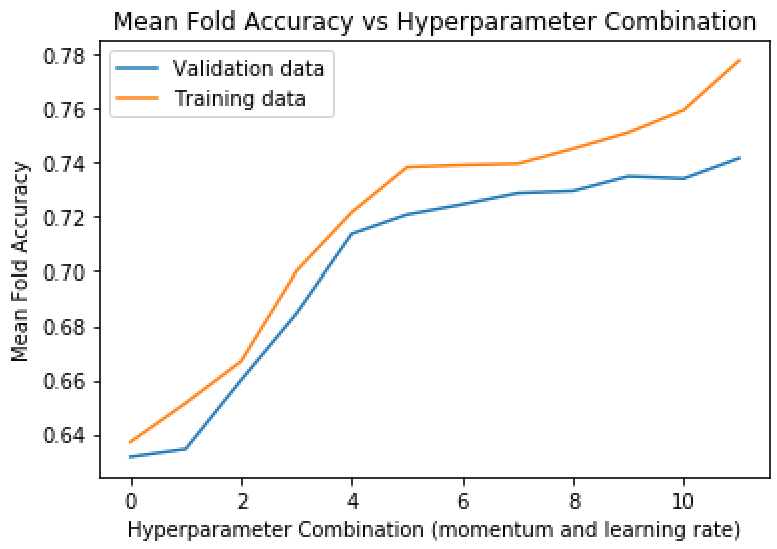
The best combination of momentum and learning rate was 0.9 for momentum and 0.01 for learning rate, with a mean validation accuracy of 77% across all cross validation runs. This was the last combination (index 11) which means that the classifier worked best when momentum and learning rate were both at their maximum available value. Standard deviation on the training set is lower for higher values of the learning rate, which is expected as higher learning_rate results in a greater weight update. For higer amounts of learning rate the model would overfit. Average testing accuracy does not fluctuate much, besides being lowest for the smallest learning rate and 0 momentum.

In [20]:

```
test_scores_mlp_we = results_mlp_we.cv_results_['mean_test_score']
train_scores_mlp_we = results_mlp_we.cv_results_['mean_train_score']

plt.plot(test_scores_mlp_we)
plt.plot(train_scores_mlp_we)
plt.title('Mean Fold Accuracy vs Hyperparameter Combination')
plt.xlabel('Hyperparameter Combination (momentum and learning rate)')
plt.ylabel('Mean Fold Accuracy')
plt.legend(('Validation data', 'Training data'))

is_positive_mlp_we = results_mlp_we.predict_proba(x_test_we)[:, 1] # Predicted p
roba values for gradescope
np.savetxt('results_mlp_we.txt', is_positive_mlp_we)
```



There is a general increasing trend in classifier accuracy for both training and validation data, the higher the momentum and learning rate is. Note that for the learning rate of 0.01 (after index 5 in x-axis) the slope is less positive than for learning rate of 0.001, but this might be the case because it is stabilizing in accuracy. For the training data, it increases gradual. We can also see that as we go across the x-axis, the difference between training and validation accuracy increases, a sign that the model starts to lose generalization powers and overfits on the training data.

2.4 Generating Third Model (SVM classifier)

For this part I will use an SVM classifier from sklearn's built in SVC class. The kernel used will be rbf, and I will evaluate the effect of C and gamma on training and testing accuracy. I will perform cross validation by splitting training data into 10 folds. The best hyperparameter combination of C and gamma will be chosen using GridSearchCV as in previous steps. C specifies the penalty for error term, and I will inspect values of [0.01, 0.1, 1, 10, 100, 1000]. These values were chosen because in literature, C is varied by factors of 10 like here so I thought it would be a good place to start. Gamma is the kernel coefficient for rbf. I will inspect these values for gamma: [0.01, 0.001, 0.0001]. There are 18 combinations for our parameters, for each of the 10 folds resulting in 180 model fits.

In [21]:

```
param_grid_svm_we ={'C': [0.01, 0.1, 1, 10, 100, 1000], 'gamma': [0.01, 0.001,
0.0001], 'kernel': ['rbf']},
results_svm_we = GridSearchCV(SVC(probability=True, kernel='rbf'), param_grid_sv
m_we, cv=10, verbose=1, n_jobs=-1, refit=True, return_train_score=True)
results_svm_we.fit(x_train_we, y_train_we)
df = pd.DataFrame(results_svm_we.cv_results_)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent w
orkers.

Fitting 10 folds for each of 18 candidates, totalling 180 fits

[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   17.8s
[Parallel(n_jobs=-1)]: Done 180 out of 180 | elapsed:  1.2min finish
ed
```

In [22]:

```
print('Best Classifier Settings: '),
print(results_svm_we.best_estimator_) # best MLPCLassifier from gridsearch
print('Best index: '),
print(results_svm_we.best_index_) # best param combo
print("Best cross-validation score: {:.2f}".format(results_svm_we.best_score_))
print('Analytical results for Standard deviation and mean fold accuracy')
tmp = pd.DataFrame(data={'C': df['param_C'], 'Gamma': df['param_gamma'], 'Std Tr
ain Dev': df['std_train_score'], 'Std Test Dev': df['std_test_score'],
                         'Mean Train Accuracy': df['mean_train_score'], 'Mean Te
st Accuracy': df['mean_test_score']})
print(tmp)
```

```
Best Classifier Settings:
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rb
f',
    max_iter=-1, probability=True, random_state=None, shrinking=Tru
e, tol=0.001,
    verbose=False)
Best index:
12
Best cross-validation score: 0.74
Analytical results for Standard deviation and mean fold accuracy
       C   Gamma  Std Train Dev  Std Test Dev  Mean Train Accuracy  \
0   0.01    0.01       0.016131      0.048992             0.703009
1   0.01   0.001       0.017446      0.038270             0.710093
2   0.01  0.0001       0.017027      0.041140             0.708519
3    0.1    0.01       0.005483      0.048153             0.710880
4    0.1   0.001       0.017446      0.038270             0.710093
5    0.1  0.0001       0.017027      0.041140             0.708519
6      1    0.01       0.004224      0.035940             0.741019
7      1   0.001       0.006332      0.043062             0.716898
8      1  0.0001       0.017027      0.041140             0.708519
9     10    0.01       0.005094      0.043623             0.752083
10    10   0.001       0.004267      0.035961             0.741111
11    10  0.0001       0.005673      0.040716             0.717500
12   100    0.01       0.004618      0.040944             0.781898
13   100   0.001       0.003945      0.040859             0.746759
14   100  0.0001       0.004301      0.036106             0.741157
15  1000    0.01       0.003399      0.039636             0.839861
16  1000   0.001       0.004626      0.043509             0.753981
17  1000  0.0001       0.004766      0.041250             0.746667

    Mean Test Accuracy
0             0.689583
1             0.705000
2             0.707917
3             0.699167
4             0.705000
5             0.707917
6             0.733333
7             0.711250
8             0.707917
9             0.727917
10            0.732917
11            0.712083
12            0.738333
13            0.730000
14            0.733750
15            0.728750
16            0.731667
17            0.730417
```

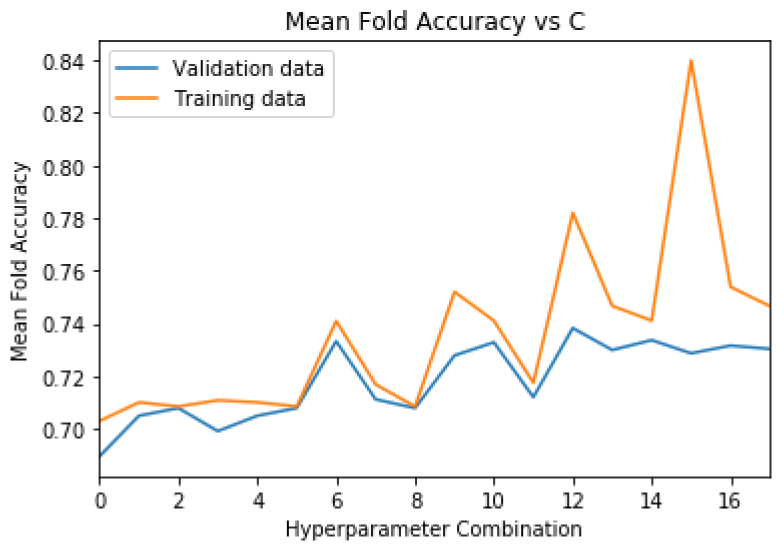
The best parameter combination with respect to validation data accuracy is for C = 1 and gamma = 0.01 with an accuracy of 76%, which is less than our best result so far.

In [23]:

```
test_scores_svm_we = results_svm_we.cv_results_['mean_test_score']
train_scores_svm_we = results_svm_we.cv_results_['mean_train_score']

plt.plot(test_scores_svm_we)
plt.plot(train_scores_svm_we)
plt.title('Mean Fold Accuracy vs C')
plt.xlabel('Hyperparameter Combination')
plt.xlim([0,17])
plt.ylabel('Mean Fold Accuracy')
plt.legend(('Validation data', 'Training data'))

is_positive_svm_we = results_svm_we.predict_proba(x_test_we)[:, 1] # Predicted p
roba values for gradescope
np.savetxt('results_svm_we.txt', is_positive_svm_we)
```



A quick look at the above graph shows high variability in accuracy by having numerous peaks. A close inspection of the above graph for our parameter pairs shows that these peaks occur for gamma=0.01. Evidence for these peaks can also be seen from the variability in standard training and testing deviation, with a higher general standard deviation for training data. As C increases, the difference in accuracy between training and validation set increases because validation accuracy stabilizes at approximately the 75% mark, while training set accuracy reaches higher values as a result of overfitting.

2.5 Best Classifier for word embeddings

The three models above had similar accuracies, with the best one being the neural network (77%). This accuracy was lower than I expected. In the feature creation pipeline the word embeddings for each word in review were simply averaged for a single 50 weight vector to represent the whole sequence. Obviously this technique fails to capture the detail needed to reach higher accuracy. This is the case because the averaging of word embeddings does not do anything to capture the semantics of the sentence as a whole. To reach accuracies higher than this level, features should also represent the semantic details of the sentence, so a more complex feature generation pipeline should be used, that will be able to capture the full power of word embeddings. I found the word embedding representation to be generally faster than the Bag of words representation which makes sense due to the lower number of features for the word embeddings (50). This allows for faster hyperparameter tuning and easier iteration to try out many different models. Compared to the SVM model tried, the neural network is better at avoiding overfitting of the data, and reaches higher accuracy with more flexibility in comparison to both the SVM and Logistic regression models.

2.6 Test best classifier on leaderboard

When the classifications of the neural network were tested against the leaderboard the results were an error rate of 0.205 and an AUROC of 0.868. As expected by our cross validation results, this model performed slightly worse than the logistic regression model with bag of words representation. The error rate of 0.205 is close to the average error rate experienced for the training and cross validation datasets (at least for lower indices in hyperparameter combinations), meaning that the model does not show any signs of drastic overfitting. With an area under ROC curve of 0.87, the number of False Positives and False Negatives are in the same caliper as in Part 1.6, with a respectable ability of distinguishing sentiment in reviews.