CLASSIFY REVIEW SENTIMENT

This is my project in Machine Learning course at Tufts University

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

1. "Pipeline" for generating BoW features

As we cannot work directly with text using machine learning algorithms, we need to convert text to numbers. The model that we are going to use is Bag-of-Word (BoW) model. This model encodes any document as a fixed-length vector with the length of the vocabulary of known words. The value in each position would be filled with a count or frequency of each word in the encoded document.

I used *CountVectorizer* to encode. It provides a simple way to tokenize a collection of text documents and build a vocabulary of known words. First, I created an instance of the *CountVectorizer* class. Second, I called the `fit_transform()` function to learn the vocabulary and to return term-document matrix for the training data set. Third, I called the `transform()` function to encode the test data set documents as vectors. This would extract token counts out of raw text documents using the vocabulary fitted from the training data set.

Results are encoded vectors with a length of the entire vocabulary and an integer count for the number of times each word appeared in the document. Below are the codes:

```
vectorizer = CountVectorizer()
X_train_counts = vectorizer.fit_transform(tr_text_list)
X_test_counts = vectorizer.transform(te_text_list)
```

Using the vectorized words, I use *"Term Frequency – Inverse Document"* frequency (TF-IDF). The motivation for this method is that some words such as "*the*" will appear many times. Their large counts will not be meaningful to the encoded vectors. The term frequency summarizes how often a given word appears within a document. The inverse document frequency downscales words that appear a lot across documents. Basically, TF-IDF highlights words that are more interesting, which are the frequent ones in a document but not across all documents.

To achieve this, I used *TfidVectorizer*, which tokenizes documents, learns the vocabulary, and inverses document frequency weightings. As I already have a learned *CountVectorizer,* I only need to use *TfidfTransformer* to calculate the inverse document frequencies and start encoding documents. Below are the codes that I used:

```
tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
X_test_tfidf = tfidf_transformer.transform(X_test_counts)
```

Now, the scores are normalized and are ready to be used directly with machine learning algorithms.

2. Generate a logistic regression model

First, I used a simple logistic regression model as a benchmark. I use `GridSearchCV()` with `LogisticRegression()` estimator. I decided to keep the parameters as default so that I can see how the change in the hyperparameter of choice varies model's performances. That means my logistic regression model has penalty = 'L2', solver = 'liblinear", and max_iter = 100. The only thing I that varied is the C value, which is the inverse of regularization strength.

The range that I chose is log space from -9 to 6 with interval = 31. Therefore, it will return numbers spaced evenly on a log scale based on the interval. It will give 31 C values, which are respective to 31 models. This number of C values is large enough to see the change in model performances if there is any. Also, this log space range is widely applied through different examples.

The cross-validation scheme is included in `GridSearchCV()`. With each hyperparameter value, the model will automatically do k-fold cross validation. I chose k = 10; therefore, we will have 310 different runs. The way cross validation works is that the training set is split to 10 smaller sets. It is trained using 9 (k-1) of the folds as training data. Then it is validated on the remaining part of the data. The performance measure is then the average of the values computed in the loop. Below are the codes, which can be very easy to reproduce:

```
param_grid = [ {'C': np.logspace(-9, 6, 31)}]

lrm = GridSearchCV(LogisticRegression(solver='liblinear'), param_grid,
cv=10, return_train_score=True ).fit(X_train_tfidf,
y_train_df.values.ravel())
```

Please note that the results for each hyperparameter value are already the average of 10-fold cross-validation. After getting the results, I decided to draw a plot to show accuracy score and another plot to show uncertainty levels (standard deviation) throughout each hyperparameter value C. The x axis is $\log_{10}$ C. For the accuracy plot, the y axis is the mean training scores and mean test scores. The scores are average accuracy of training and validation sets after doing 10-fold cross validation tests. For the standard deviation plot, the y axis is the standard deviation across 10-fold cross validation tests.

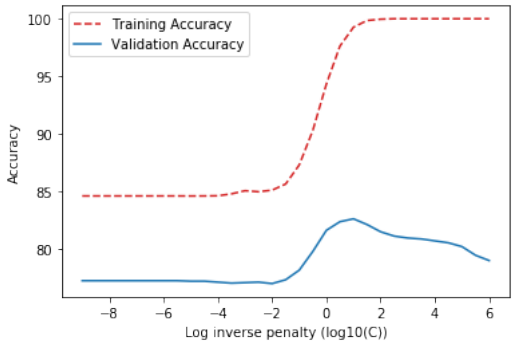


Figure 1: Accuracy score for Logistic clf

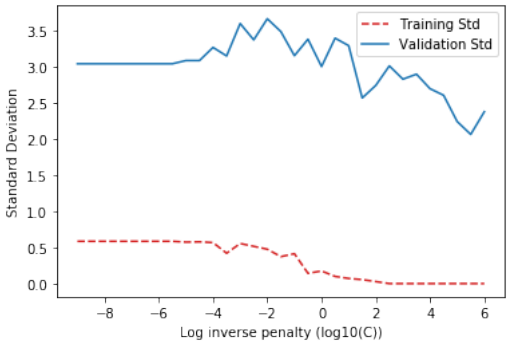


Figure 2: Standard deviation for Logistic clf

For accuracy, there are hyperparameter settings for which the classifier clearly does better. Overall, there are significant differences between training accuracy and validation accuracy. From

-8 to 0, there is approximately a 5% accuracy difference. However, from 0 onwards, training accuracy surged to nearly 100% while validation accuracy only slightly increased. As there are clear differences in performances between training and validation sets, there may be evidence of over-fitting from the value log10(C) = 0.

However, for uncertainty, standard deviation of training set is much lower than standard deviation of validation test. Therefore, we can see that training set has much lower variance across different log inverse penalty values, in which we can conclude that overfitting may happen.

3. Generate a MLP model

Now, I will train the model using MLP. I use `GridSearchCV()` with `MLPClassifier()` estimator. For the parameters, I set activation function = 'logistic'. I choose L-BFGS optimization function, which uses both the first and second derivatives. I expect that this optimization function will yield higher accuracy compared with SGD.

The hyperparameter that I will choose to make variations is alpha, which is the regularization term aka penalty term. Alpha handles overfitting by constraining the size of the weights. Increasing alpha means fixing high variance, which is a sign of overfitting. Lowering alpha means fixing high bias, which is a sign of underfitting.

My alpha values are log space range from -5 to 3 with 20 values in between. Again, this is a popular range used in multiple examples. I chose the number 20 so that we are able to see enough variations in performance across values if there is any without affecting too much training time.

Similar to the previous model, I chose k = 10 in k-fold cross-validation. Here are the codes, which are very easy to reproduce:

```
param_mlp = [ {'alpha': np.logspace(-5, 3, 20)} ]

mlpclf = GridSearchCV(MLPClassifier(activation = 'logistic',
solver='lbfgs'), param_mlp, cv=10, return_train_score=True
).fit(X_train_tfidf, y_train_df.values.ravel())
```

Similarly, I will print 2 plots, one for accuracy and one for standard deviation.

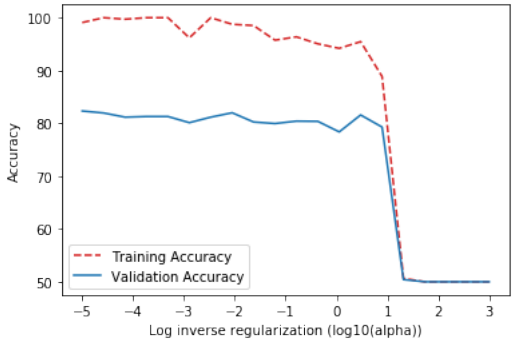


Figure 3: Accuracy for MLP

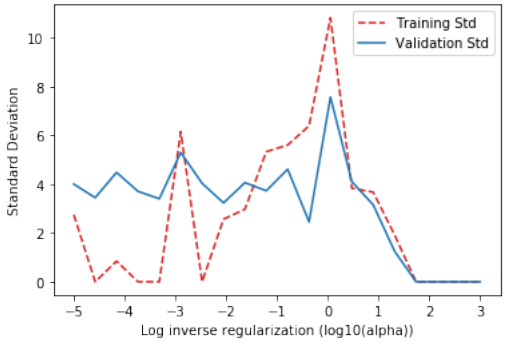


Figure 4: Standard deviation for MLP

It is interesting to see that after a few alpha values, accuracy scores of both training and validation sets reduced significantly. For the first few, it looks like varying the alpha values do not

effectively increase the accuracy, as the accuracy scores for both training and validation sets are quite stable. Looking at the standard deviation plot, it is very noisy. Therefore, we cannot conclude about the signs of overfitting in this case.

4. Generate a SVM model

The third model that I chose is SVM classifiers. For the parameters, I chose kernel = 'rbf' and gamma = 'scale', as the data that we have is imbalanced.

The hyperparameter that I chose is C, which is penalty parameter of the error term. For larger value of C, the optimization will choose a smaller-margin hyperplane. For a smaller value of C, the optimizer will look for a larger-margin separating hyperplane, even if the hyperplane misclassifies more points. I used a log space range for C from -2 to 10 with 13 C values.

I chose k = 10 in k-fold cross-validation. Here are the codes:

```
param_svm = [ {'C': np.logspace(-2, 10, 13)} ]

svmclf = GridSearchCV(SVC(kernel='rbf', gamma = 'scale'), param_svm,
cv=10, return_train_score=True ).fit(X_train_tfidf,
y_train_df.values.ravel())
```

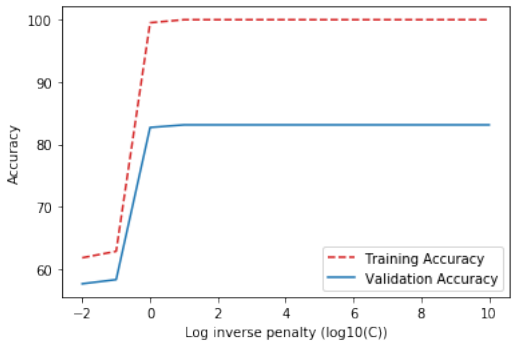


Figure 5: Accuracy score for SVM

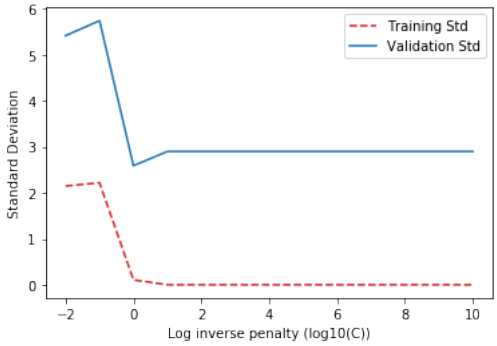
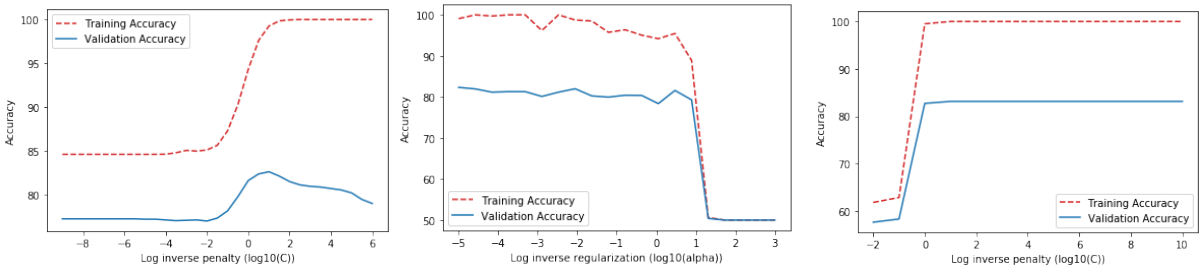


Figure 6: Standard deviation for SVM

Regarding accuracy, it was very low in the first few C values, then increased significantly. The validation accuracy looks pretty good, at 80%. We can say that as C values increased, the classifier did better. In terms of standard deviation, training set has much lower variance; therefore, we can conclude that overfitting may happen.

5. Summarize classifier performances

Let's compare performances of 3 classifiers that we did, with the order is Logistic – MLP – SVM:



We can see that among 3 classifiers, the first one, which is Logistic regression, has the most stable performance. Looking at validation set accuracy, it always has approx. 80% accuracy, while for the remaining 2 classifiers, the scores are volatile either with later variations in hyperparameters or with early variations in hyperparameters. However, logistic classifier here does not really outperform SVM. Overall, SVM has very good validation accuracy as well. No classifier shows clear advantages in preventing overfitting, since for logistic and SVM, training accuracy is still much higher than validation accuracy. For MLP, before the log10(alpha) value reaches 1, training accuracy is still much higher than validation accuracy too.

The hypothesis is that logistic regression has very few parameters compared with MLP and SVM. However, probably since classifiers in this context do not have to represent XOR, logistic regression still shows its advantages. Otherwise, it may not be the best classifier.

Here are a few examples of the False Positives:

```
"I've had this bluetoooth headset for some time now and still not comforta
ble with the way it fits on the ear."

"I tried talking real loud but shouting on the telephone gets old and I wa
s still told it wasn't great."

'Problem is that the ear loops are made of weak material and break easily.
'

'Adapter does not provide enough charging current.'
```

First, all of those sentences are from Amazon. Second, the common mistake is that the classifier misses such negative words as “still not”, “wasn’t”, and “does not”. It only recognizes the positive words such as “comfortable”, “great”, and “enough”. For the 3rd document, it only gets the word “easily” and thinks that is a positive review. It’s indeed really hard for the classifier to realize “break easily” means bad.

Here are a few examples of the False Negatives:

```
'the only VERY DISAPPOINTING thing was there was NO SPEAKERPHONE!!!!'

'It is cheap, and it feel and look just as cheap.'
```

Again, all of these are from Amazon. The mistake is quite similar: the classifier recognizes “very disappointing” as bad and labels as negative review; however, “the only very disappointing…” might mean positive, as the product is good overall. For the second review, it’s a bit confusing why it is actually a positive review. “Cheap” might normally be classified as bad.

6. Compare classifier validation performances with test performances

Classifier	Error Rate	AUROC
Logistic Regression	0.19167	0.80833
MLP	0.245	0.755
SVM	0.415	0.585

These results are from the leader board. It is clear that logistic regression earned the highest performance with lowest error rate and highest AUROC. This matches with the expectation and with the validation set performance.

Looking at these results, we cannot say one classifier absolutely outperforms another. This is because the performance depends a lot on hyperparameters and parameters. Probably with other parameter variations, MLP and/or SVM may yield higher accuracy than LR.

Part Two: Classifying Review Sentiment with Word Embeddings

1. Feature-generation pipeline from GloVe

First, I split the training data into words by blank space. Then, I used `nltk.tokenize.RegexpTokenizer` to tokenize, transforming each document into a list divided by individual words. All words are in lower case.

Here is a snapshot:

```
[it, always, cuts, out, and, makes, a, beep, b...
[the, only, very, disappointing, thing, was, t...
```

After that, I decided what to do with typos. For words that are not included in the dictionary word2vec, it will return 'None'. Next, I created vectorized values to fit into the models. The idea is that each word in a document will yield a vector 1x50 when passing into word2vec. Then, I decided to **average** the values of individual word vectors. Decisions about what I should do when a word appears many times or should I ignore rare or common words would be explored further in the future. Here are my codes to preprocess data into the form that can be fitted into models:

```
def word2vec2(word):
    try:
        return(word2vec[word])
    except:
        return None
```

```
def sen2vec(sen):
    # sen: a list of words
    l1 = [word2vec2(item) for item in sen]
    l1 = [item for item in l1 if item is not None]
    return [np.mean([item[i] for item in l1]) for i in range(50)]
```

```
tr_data = [sen2vec(sen) for sen in train2]
```

2. Generate a logistic regression model

Similar to the BoW model, I use `GridSearchCV()` with `LogisticRegression()` estimator. The parameters are the same to the previous part's logistic regression model, in which penalty = 'L2', solver = 'liblinear", and max_iter = 100.

Regarding hyperparameter C, I decided to vary in a log space range from -9 to 6 with interval = 20 (I want to have similar intervals across all models, and I choose the number of intervals is 20).

The cross-validation scheme is included in `GridSearchCV()`. With each hyperparameter value, the model will automatically do k-fold cross validation. I chose k = 10; therefore, we will have 200 different runs. The way cross validation works is that the training set is split to 10 smaller sets. It is trained using 9 (k-1) of the folds as training data. Then it is validated on the remaining part of the data. The performance measure is then the average of the values computed in the loop. Below are the codes, which can be very easy to reproduce:

```
param_grid = [{'C': np.logspace(-9, 6, 20)}]

lrm = GridSearchCV(LogisticRegression(solver='liblinear'), param_grid,
cv=10, return_train_score=True ).fit(tr_data,
y_train_df.values.ravel())
```

After getting the results, I made two plots showing accuracy and standard deviation.

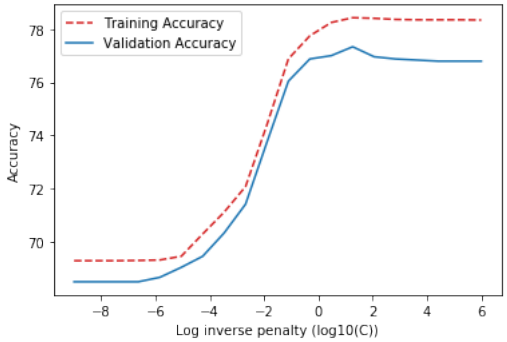


Figure 7: Accuracy for Logistic regression

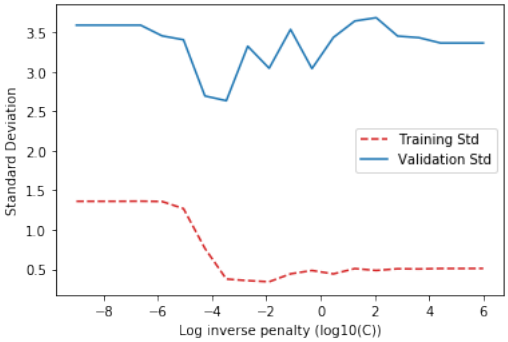


Figure 8: Standard deviation for Logistic regression

We can see that after approximately half of the range C values, the accuracy increased significantly from around 70% to nearly 80%. This means certain higher C values lead to higher accuracy. After the value of 0 for log inverse penalty, the accuracy becomes stable. Accuracy also does not necessarily correlate with higher number of intervals. I tried with number of intervals = 50, the accuracy improved by a very tiny margin.

In terms of uncertainty, training data has much lower standard deviation compared with validation set. Therefore, we can see that as a sign of overfitting.

3. Generate a MLP model

Now, I will train the model using MLP. I use `GridSearchCV()` with `MLPClassifier()` estimator. For the parameters, I set activation function = 'logistic'. I choose L-BFGS optimization function, which uses both the first and second derivatives. I expect that this optimization function will yield higher accuracy compared with SGD.

The hyperparameter that I will choose to make variations is alpha. My alpha values are log space range from -5 to 3 with 20 values in between. I chose k = 10 in k-fold cross-validation. Here are the codes, which are very easy to reproduce:

```
param_mlp = [ {'alpha': np.logspace(-5, 3, 20)} ]
```

```
mlpclf = GridSearchCV(MLPClassifier(activation = 'logistic',
solver='lbfgs'), param_mlp, cv=10, return_train_score=True
).fit(X_train_tfidf, y_train_df.values.ravel())
```

Similarly, I will print 2 plots, one for accuracy and one for standard deviation.

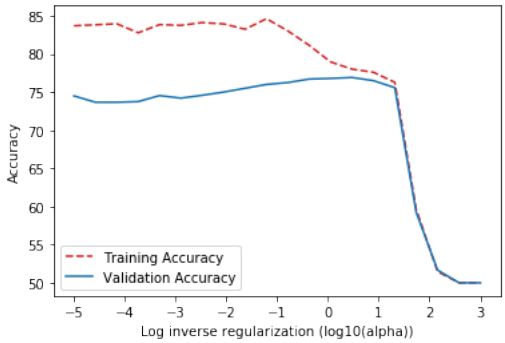


Figure 9: Accuracy for MLP

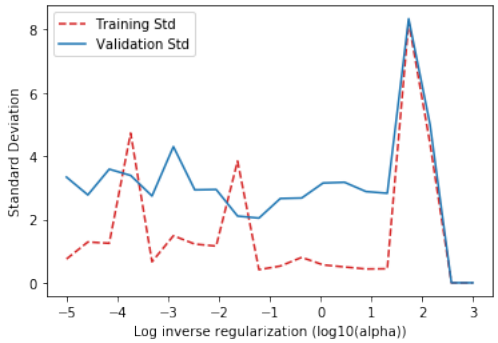


Figure 10: Standard deviation for MLP

In terms of accuracy, there is a quite clear difference between training and validation data's accuracies. From the log inverse regularization value of -5 to around 1.6, training accuracy is within the range from 76% to 85%. However, validation accuracy is stable around 75%. The similarity between training and validation data is that after the log inverse regularization value of around 1.6, accuracy reduces dramatically to around 50%, which makes MLP classifier now becomes a weak learner.

In terms of standard deviation, the only match between training and validation data is when the log inverse regularization alpha value reaches 2. For other values, standard deviation is very fluctuating. For most parts, validation standard deviation is higher than training standard deviation, which is quite a sign of overfitting.

4. Generate a SVM model

For the third model, I still choose SVM classifiers. I chose kernel = 'rbf' and gamma = 'scale'. The hyperparameter that I chose is C. I used a log space range for C from -2 to 10 with 20 C values.

I chose k = 10 in k-fold cross-validation. Here are the codes:

```
param_svm = [{'C': np.logspace(-2, 10, 20)}]
svmclf = GridSearchCV(SVC(kernel='rbf', gamma = 'scale'), param_svm,
cv=10, return_train_score=True ).fit(tr_data,
y_train_df.values.ravel())
```

Here are the plots for accuracy and standard deviation:

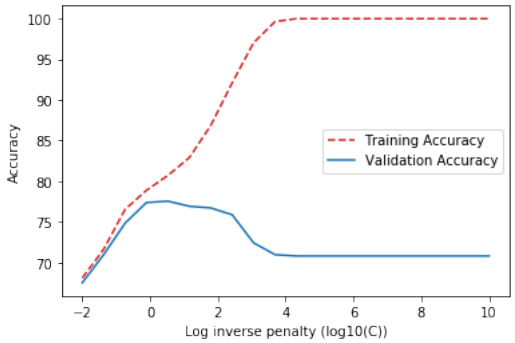


Figure 11: Accuracy for SVM

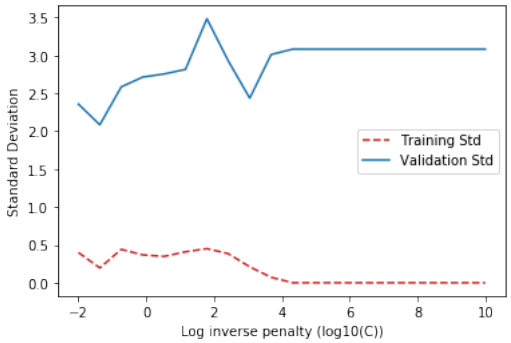


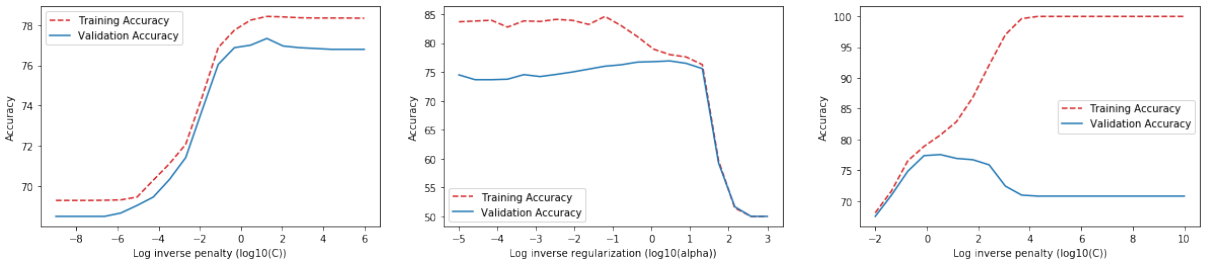
Figure 12: Standard deviation for SVM

Regarding accuracy, from the log inverse penalty value of -2 to 2, there is not much of the differences between accuracy of training and validation data sets. However, after the value of 2, there is a clear difference. Validation accuracy is still around 72%, while training accuracy is nearly 100%. We can see a clear example of overfitting here.

Regarding standard deviation, it is confirmed that there is clear evidence of overfitting, as validation error is very high compared with training error.

5. Summarize classifier performances

Let's compare performances of 3 classifiers that we did, with the order is Logistic – MLP – SVM:



Overall, logistic regression classifiers performed the best, even it has a lower starting point. Until the last few hyperparameter values, logistic regression's validation accuracy is quite matched with its training accuracy, which is a sign that this classifier can reduce overfitting.

Here are a few False Positives:

```
'Not impressed.'

"I've had this bluetoooth headset for some time now and still not comforta
ble with the way it fits on the ear."

'dont buy it.'

'I have had this phone for over a year now, and I will tell you, its not t
hat great.'

'I can barely ever hear on it and am constantly saying "what?"'
```

First, all of those sentences are from Amazon. Second, as the last part, the common mistake is that the classifier misses such negative words as "still not", "don't", "it's not", and "barely". It only recognizes the positive words such as "comfortable", "buy", and "great". For the 3rd document, it only gets the word "can [...] hear" and thinks that is a positive review. It's indeed really hard for the classifier to realize "can barely ever hear" means bad.

Here are a few False Negatives:

```
'It is cheap, and it feel and look just as cheap.'

'the only VERY DISAPPOINTING thing was there was NO SPEAKERPHONE!!!!'

"Doesn't hold charge."

'Pretty piece of junk.'
```

This is quite similar to the FNs in BoW. However, there are reviews that are just negative, which the classifier does it correctly. Probably there may be issues with the labels.

6. Compare classifier validation performances with test performances

Classifier	Error Rate	AUROC
Logistic Regression	0.26833	0.73167
MLP	0.26833	0.73167
SVM	0.26167	0.73833

These results are taken after uploading to the leader board. It is really interesting to see that logistic regression has exact error rate and AUROC as MLP. Given the two hyperparameter ranges and similar number of hyperparameter values, there is no difference in performances between these 2 models.

We can clearly see here SVM is the best model out of 3. The hypothesis is that SVM works well with unstructured data such as text and images. It uses the kernel trick, which is a real strength of SVM. Also, unlike neural networks, SVM does not solve for local optima. SVM also can scale relatively well to high dimensional data.