Project 1
Classification Analysis on Textual Data

Donna Branchevsky
UID: 404473772

Pavan Holur
UID: 204403134

Megan Williams
UID: 104478182

Tadi Ravi Teja Reddy
UID: 505227246

University of California, Los Angeles

Winter 2019

Question 1

Importing the news groups from the scikit-learn library into python's working directory, we observed the 20 classes of training data. The classes were labelled by context, with the training data modelled as text snippets associated with each label. The following code examined the 20 classes as a frequency measure:

```python
def question1(flag = True):
  newsgroups_train = fetch_20newsgroups(subset='train')
  plt.hist(newsgroups_train['target'], bins=80)
```

Adjusting plot settings we get the histogram below. This plot represents the number of articles vs. class label to be a roughly balanced distribution; there was reasonably equal representation for all classes in the data, especially those we examined in later context.

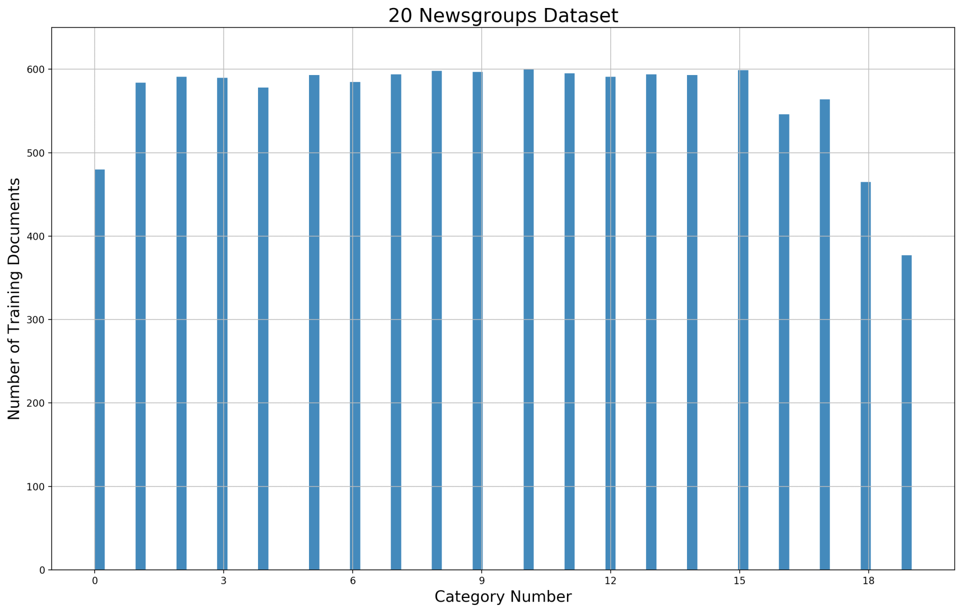

Figure 1: Histogram for data into 20 context bins

Question 2

Before proceeding, the random seed was fixed to 42, so that all analyses were reproducible, and all random processes were now pseudo-random. The categories extracted from the data set for binary classification were sourced as follows:

1. Among the 20 classes, 8 classes were selected and grouped into two broader classes. The eight sub classes were: 'comp.graphics', 'comp.os.ms-windows.misc',

’comp.sys.ibm.pc.hardware’, ’comp.sys.mac.hardware’, ’rec.autos’, ’rec.motorcycles’, ’rec.sport.baseball’, ’rec.sport.hockey’.

2. The first 4 and last 4 classes unionized to broader classes named ’Computer Technology’ and ’Recreational Activity’ respectively.

This process resulted in 2 large class labels comprising of 4 base labels each. Note from Question 1 that the two super-labels formed had balanced data sets. This process was accomplished by using the *fetch_20newsgroups* function again, with specified categories from above. Once obtained, the features used for each data point was a ”Bag of Words” model, which counts the frequency of words in a sample data point. Smart tools from the NLTK library allowed us to extract unique features for words with a) similar stems b) varied tenses c) non-stop words, and d) non-digits. To create such a feature extractor, an object of *CountVectorizer* was created (see below):

```python
def vectorize_data(categories, min_df=3, binary=True, print_stop_words = False):
    train_dataset = fetch_20newsgroups(subset = 'train', categories = categories, shuffle = True, random_state = None)
    test_dataset = fetch_20newsgroups(subset = 'test', categories = categories, shuffle = True, random_state = None)

    if binary:
        Bin_Target_Train = [int(i > 3) for i in train_dataset.target]
        Bin_Target_Test = [int(i > 3) for i in test_dataset.target]
    else:
        Bin_Target_Train = train_dataset.target
        Bin_Target_Test = test_dataset.target

    count_vect = build_vectorizer(min_df=min_df, print_stop_words=print_stop_words)

    X_train_counts = count_vect.fit_transform(train_dataset.data)
    X_test_counts = count_vect.transform(test_dataset.data)
```

Once the large feature vectors have been formed, we ignored features that were passed by the *CountVectorizer*, but still were not distinguishing factors between the two binary classes, because of their contextual dominance in overpowering our classification. Note that the min_df setting in the *CountVectorizer* ignored the noise in the text data by avoiding bags for which the word count was < 3. To avoid the bias, we further constructed the TF-IDF vectors: normalized vectors that removed implicit offset in the data. We constructed the transformed vectors as shown below:

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

The function *vectorize_data* then returns X_train_tfidf, X_test_tfidf, Bin_Target_Train, Bin_Target_Test. The bag-of-words features from the Count Vectorizer were now normalized through the TF-IDF transform, yielding training and test vectors. Note that the test data was bagged with trained model obtained from the training data. The resulting training and test data matrix dimensions were: (the target variable is simply a 1D vector with the binary target values).

```
$ python Project_1_q1_q2.py
('X_train_tfidf size: ', (4732, 16292))
('X_test_tfidf  size: ', (3150, 16292))
```

Question 3

The dimensions of the data above was denoted with (x,y) where x is the number of documents and y is the number of features. It was important to verify that the number of features in training and test data are identical (after all the dictionary in the *CountVectorizer* was training from the training data only).

Observing that the number of features in the data was still too large, the validity of our classifiers was questionable as the feature space was exponentially complex to classify. We sought to reduce the dimensions of our feature vectors in two ways: a) Latent Semantic Indexing (LSI), and b) Non-negative Matrix Factorization (NMF).

- In LSI, the Singular Value Decomposition (SVD) of the training matrix X_train was found and the top n eigenvalues were recovered. In the expression $X_{train} = U\Sigma V^T$, we obtained a reduced $X_{train,r} = X_{train}V_n$ in which all data vectors are only 'n' length. Also the test data was projected onto the same V basis to yield $X_{train,r} = X_{test}V_n$.

- NMF uses an optimization of the form $min\ \|(X_{train} - W_{train}H_{fixed})\|$. While W_{train} is the reduced feature set for the training data, the test features are transformed using H_{fixed} as another optimization, $min\ \|(X_{test} - W_{test}H_{fixed})\|$.

The code to reduce dimensions was similar in LSI and NMF. The function for NMF is given below:

```
def NMF_(X_train_tfidf, X_test_tfidf):
    model = NMF(n_components=50, init='random', random_state=0)

    W_train_r = model.fit_transform(X_train_tfidf)
    W_test_r = model.transform(X_test_tfidf)

    H = model.components_
    Err_NMF = 0
    Err_NMF = np.sum(np.array(X_train_tfidf - W_train_r.dot(H))**2)
    return W_train_r, W_test_r, H, Err_NMF
```

Note that both W and H are non-negative matrices and the rows of H denote topics while its columns, the importance of each bag of words to that topic. In some sense

H is the model and W is the reduced matrix. The errors of the two approaches with $k = 50$ yielded:

```
$ python Project_1_q3.py
('Error in LSI (squared): ', 4107.9653)
('Error in NMF (squared): ', 4146.2007)
```

Applying LSI and NMF to the data, we found that LSI performed categorically better than NMF. This is because TF-IDF data X_{train} has negative values and the NMF method cannot assign negative parameters to either W or H. However, LSI is more flexible in modelling and can reach a lower error bound.

Question 4

Support vector machines (SVM) is a supervised machine learning algorithm which is used for classification and regression problems. We plot our data in an n-dimensional space (n is the number of features), and use the algorithm to find the hyper-plane that differentiates the classes.

Linear Support vector machines aim to learn a vector of feature weights,w, and an intercept, b, given the training data set. Once the weights are learned, the label of a data point is determined by thresholding $(w^T x + b)$ with 0, i.e. $sign(w^T x + b)$. Alternatively, one produce probabilities that the data point belongs to either class, by applying a logistic function instead of hard thresholding, i.e. calculating $(w^T x+b)$. w and b are learnt by solving the following optimization problem:

$$\underset{w,b}{min} \frac{1}{2}\|w\|^2 + \gamma \sum_{i=1}^{n} \xi_i \tag{1}$$

$$s.t\ y_i(w^T x + b) \geq \xi_i \tag{2}$$

$$\xi_i \geq 0 \tag{3}$$

$\forall i \in \{1, 2, 3, ...n\}$. Here, x_i is the i^{th} data point, and $y_i \in \{-1, +1\}$ is the class label and the trade-off parameter γ controls relative importance of misclassification and margin. We tested the dataset with both, soft ($\gamma = 0.0001$) and hard ($\gamma = 1000$) margin.

We first look at Receiver operating characteristic curve (or ROC curve), which is a plot of the true positive rate (TPR) against the false positive rate (FPR) and it represents the trade-off between sensitivity and specificity. Also, the closer the curve follows the left-hand border and then the top border of the ROC space, the more accurate the test.

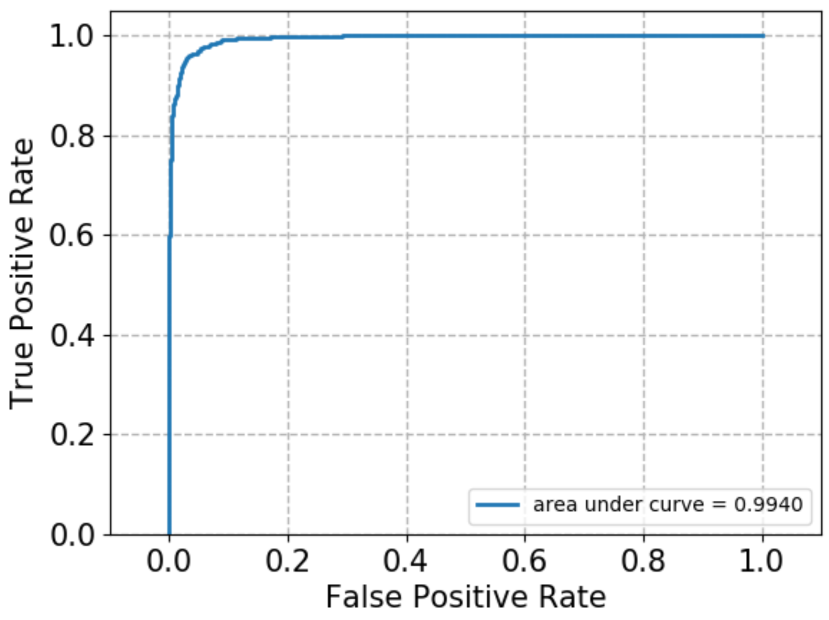

Figure 2: ROC for SVM with $\gamma = 0.0001$

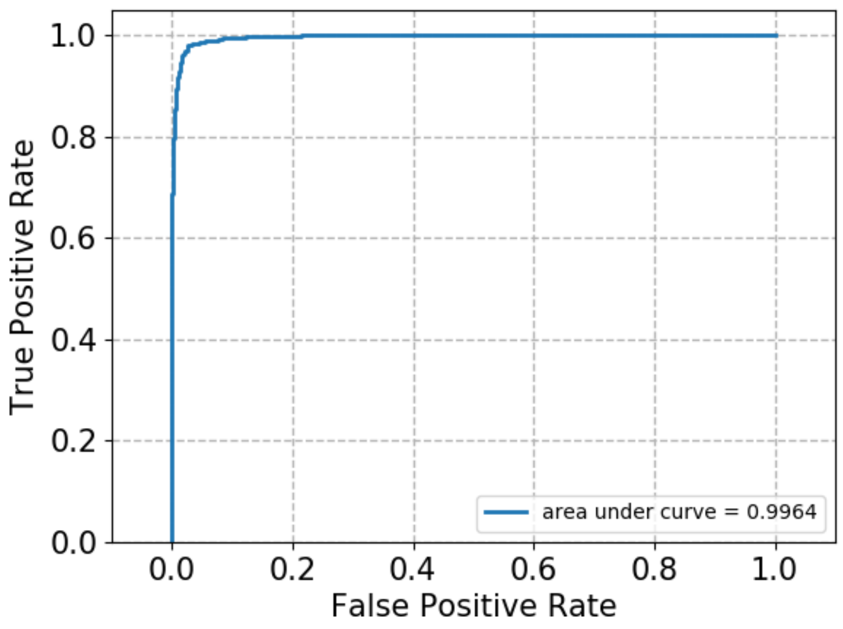

Figure 3: ROC for SVM with $\gamma = 1000$

Note: The area under the curve is a measure of text accuracy. From the ROC plots of hard and soft margin, we notice that hard margin gives better accuracy in comparison to soft margin, even though there's only a subtle difference.

We next look at confusion matrices, a confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and are broken down by each class. The confusion matrix help us identify the ways in which classification model is confused when it makes predictions.
From the confusion matrix we can compute the following metrics for our binary classification problem to evaluate the performance of SVM's:

1. Accuracy: $\frac{(Truepositives+Truenegative)}{Totalexamples}$
2. Precision: $\frac{(Truepositives)}{(Truepositives+Falsepositives)}$
3. Recall: $\frac{(Truepositives)}{(Truepositives+Falsenegatives)}$
4. F1-score: $2 \times \frac{(Precision \times Recall)}{(Precision+Recall)}$

From the confusion matrix it is clear that SVM with hard margin performs better than the SVM with soft margin. We compute accuracy, precision recall and F-1 score for SVM with $\gamma = 0.0001$ and $\gamma = 1000$.

```
print('SVM with Gamma = 0.0001')
print('Accuracy: ', accuracy_score(Bin_Target_Test, Bin_Target_predict2))
print('Precision: ', precision_score(Bin_Target_Test, Bin_Target_predict2))
print('Recall: ', recall_score(Bin_Target_Test, Bin_Target_predict2))
print('F1-score: ', f1_score(Bin_Target_Test, Bin_Target_predict2))

print('SVM with Gamma = 1000')
print('Accuracy: ', accuracy_score(Bin_Target_Test, Bin_Target_predict1))
print('Precision: ', precision_score(Bin_Target_Test, Bin_Target_predict1))
print('Recall: ', recall_score(Bin_Target_Test, Bin_Target_predict1))
print('F1-score: ', f1_score(Bin_Target_Test, Bin_Target_predict1))
```

We got the following results:

```
SVM with Gamma = 0.0001
Accuracy:  0.6723809523809524
Precision:  0.6064073226544623
Recall:  1.0
F1-score:  0.754985754985755
SVM with Gamma = 1000
Accuracy:  0.9511111111111111
Precision:  0.9155092592592593
Recall:  0.9949685534591195
F1-score:  0.953586497890295
```

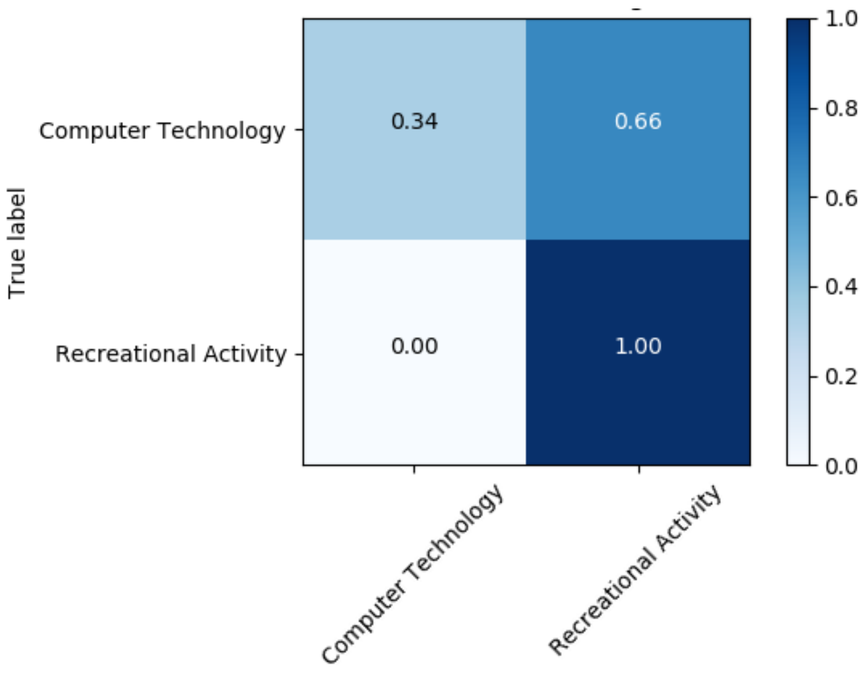

Figure 4: Confusion Matrix for SVM with $\gamma = 0.0001$

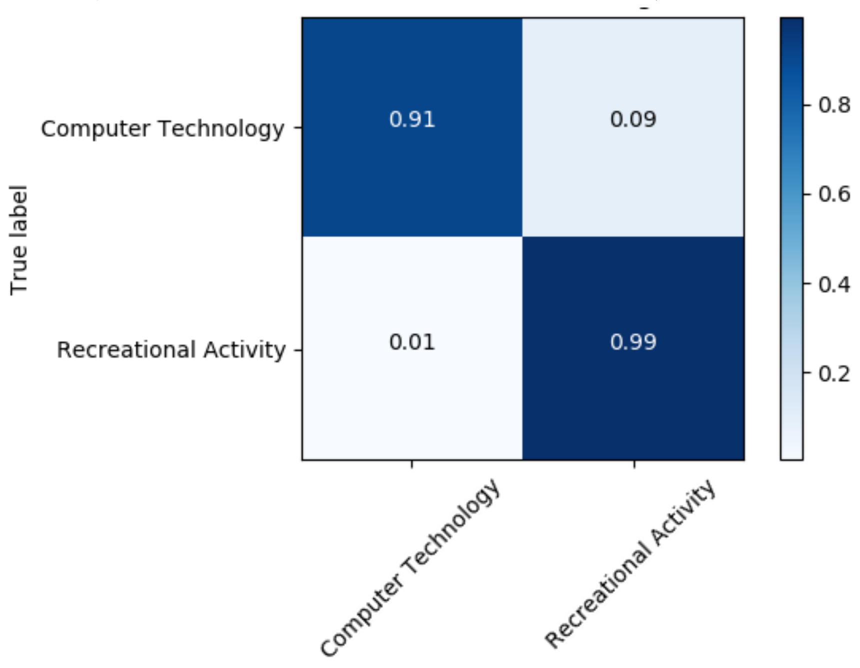

Figure 5: Confusion Matrix for SVM with $\gamma = 1000$

We observe that the inference drawn from confusion matrix coincides with ROC plots, both point that hard margin SVM's are better than soft margin SVM's. Note that the area under the curve for ROC of $\gamma = 0.0001$ does not tally well with the accuracy obtained from confusion matrix.

Next, we try to find the best γ value for our data set, we accomplish this by performing a 5-fold cross-validation on our data set for γ in the range of $\{10^k | -3 \leq k \leq +3,\ k \in \mathbb{Z}\}$ and compare various metrics to decide the best γ.

```
kfold = 5

model_0 = LinearSVC(C=0.0001)

scoring = {'accuracy': make_scorer(accuracy_score),
           'precision': make_scorer(precision_score),
           'recall': make_scorer(recall_score),
           'f1_score': make_scorer(f1_score)}
results_m0 = {}

for method in enumerate(scoring):
    results_m0[method[1]] = cross_val_score(estimator=model_0,
                                            X=X_train_r,
                                            y=Bin_Target_Train,
                                            cv=kfold,
                                            scoring=scoring[method
    [1]])

print('SVC with gamma= ', 0.0001, 'Results: ')
print("results_accuracy:  ", np.mean(results_m0['accuracy']))
print("results_precision: ", np.mean(results_m0['precision']))
print("results_recall:    ", np.mean(results_m0['recall']))
print("results_f1_score:  ", np.mean(results_m0['f1_score']))
```

We repeat the code for aforementioned values of γ and got the following results:

```
SVC with gamma = 0.0001 Results:
results_accuracy:   0.648142525825294
results_precision:  0.589338554618583
results_recall:     1.0
results_f1_score:   0.7416019833605009
SVC with gamma = 0.001 Results:
results_accuracy:   0.9397737366461122
results_precision:  0.8986432863084197
results_recall:     0.9928835206091069
results_f1_score:   0.9433681532417566
SVC with gamma = 0.01 Results:
results_accuracy:   0.9683000137055784
results_precision:  0.9627969427965313
results_recall:     0.9748839942808522
results_f1_score:   0.9687965574641234
SVC with gamma = 0.1 Results:
```

```
results_accuracy:    0.972949392396852
results_precision:   0.972820744912221
results_recall:      0.9736287641553292
results_f1_score:    0.9732086413439239
SVC with gamma = 1 Results:
results_accuracy:    0.9756957984801111
results_precision:   0.9769401440771114
results_recall:      0.9748848714507513
results_f1_score:    0.9759006288938019
SVC with gamma = 10 Results:
results_accuracy:    0.9771770562404388
results_precision:   0.9798110517258569
results_recall:      0.9748866257905494
results_f1_score:    0.9773350181495468
SVC with gamma = 100 Results:
results_accuracy:    0.9750640066339582
results_precision:   0.9769297035746799
results_recall:      0.9736313956650264
results_f1_score:    0.9748516093492402
SVC with gamma = 1000 Results:
results_accuracy:    0.9746400555196999
results_precision:   0.9835039148228759
results_recall:      0.946851398647404
results_f1_score:    0.9697296749476305
```

It's clear from the results that the optimal value of γ is 10. We plot the ROC and confusion matrix for $\gamma = 10$ and compare it with results for $\gamma = 0.0001$ and $\gamma = 1000$.

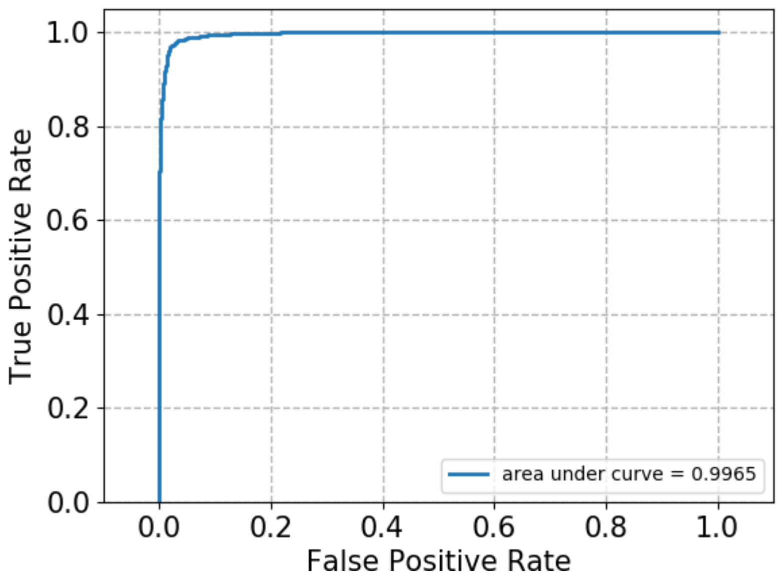

Figure 6: ROC for SVM with $\gamma = 10$

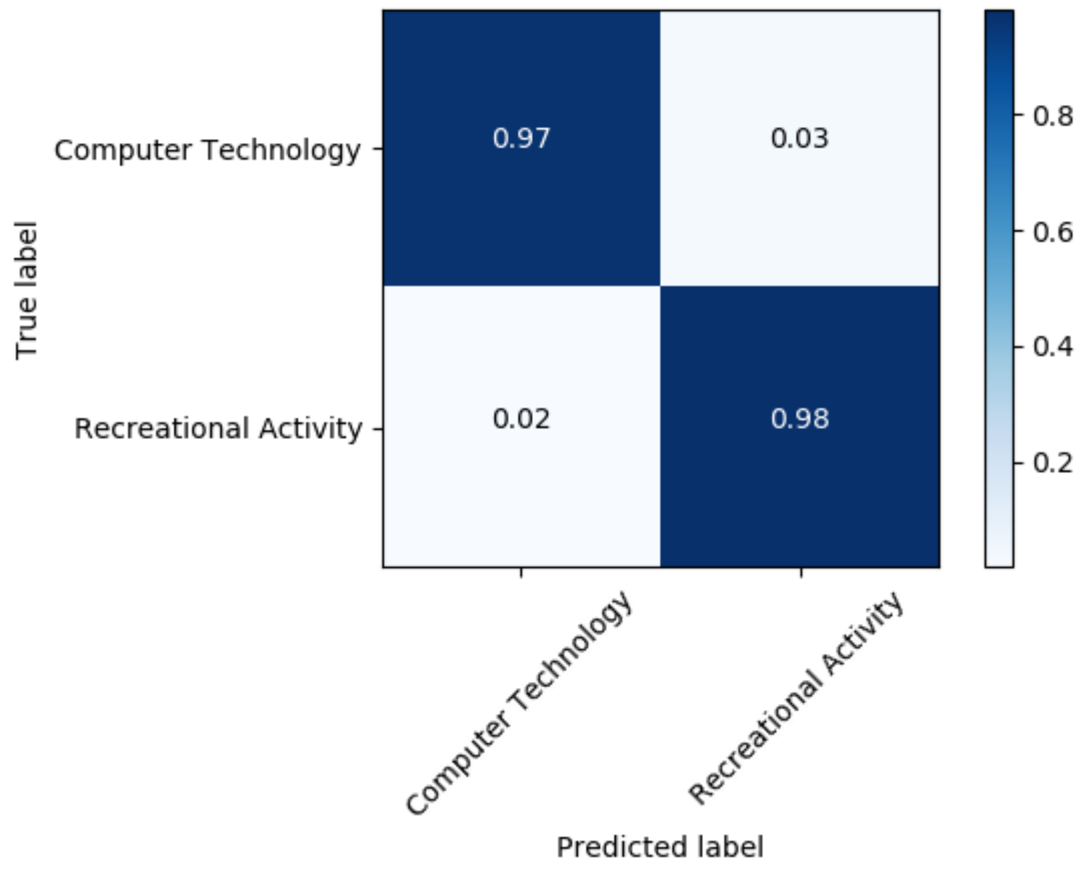

Figure 7: Confusion Matrix for SVM with $\gamma = 10$

From the ROC plots and confusion matrix, we see that SVM with $\gamma = 10$ performs best with respect to all the parameters. The accuracy, presicion, recall and f-1 scores for $\gamma = 10$ are as follows:

```
SVC with gamma = 10 Results:
 results_accuracy:    0.9771770562404388
 results_precision:   0.9798110517258569
 results_recall:      0.9748866257905494
 results_f1_score:    0.9773350181495468
```

Question 5

In logistic regression we attempt to learn a classifier $h_\theta(x) = \sigma(w^T x + b)$, where $\sigma(\phi) = 1/(1 + e^{-\phi})$. We do this by maximizing the log-likelihood of the data,

$$l(\theta) = \sum_{i=1}^{n} y_i \log h_\theta(x) + (1 - y_i) \log(1 - h_\theta(x))$$

This differs from SVM because in SVM we are attempting to find a decision boundary that maximizes the margin between data, whereas with logistic regression we want a decision boundary that maximizes the likelihood of the data.

We trained logistic classifiers using no regularization, L1 regularization, and L2 regularization. We used scikit-learn's `LogisticRegression` model, but encountered an issue because there is no way to disable regularization with this function. To get around this issue we set the C parameter to a large value, using the following code:

```
c = 1e30 # Approximate no regularization with large C
clf = LogisticRegression(C=c, solver='liblinear').fit(X,y)
```

In scikit-learn's `LogisticRegression` function, C represents the inverse of the regularization strength, so making C very large effectively nullifies the regularization.

We trained the logistic classifier using the dimension-reduced data from LSI. The ROC curve and confusion matrix for the logistic classifier run on the test data are shown below.

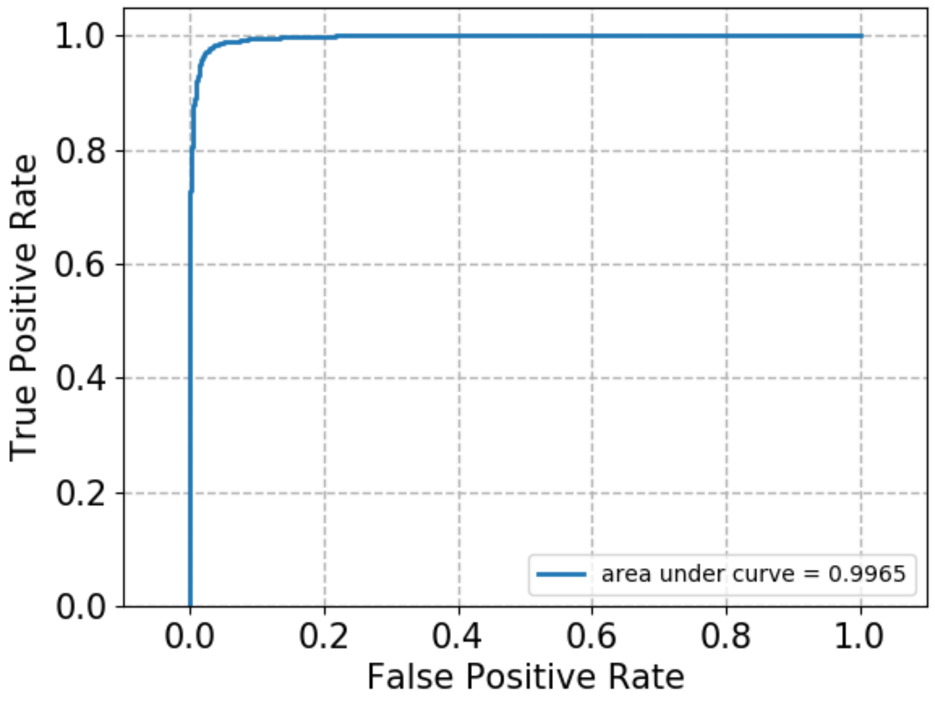

Figure 8: ROC curve for logistic classifier with no regularization

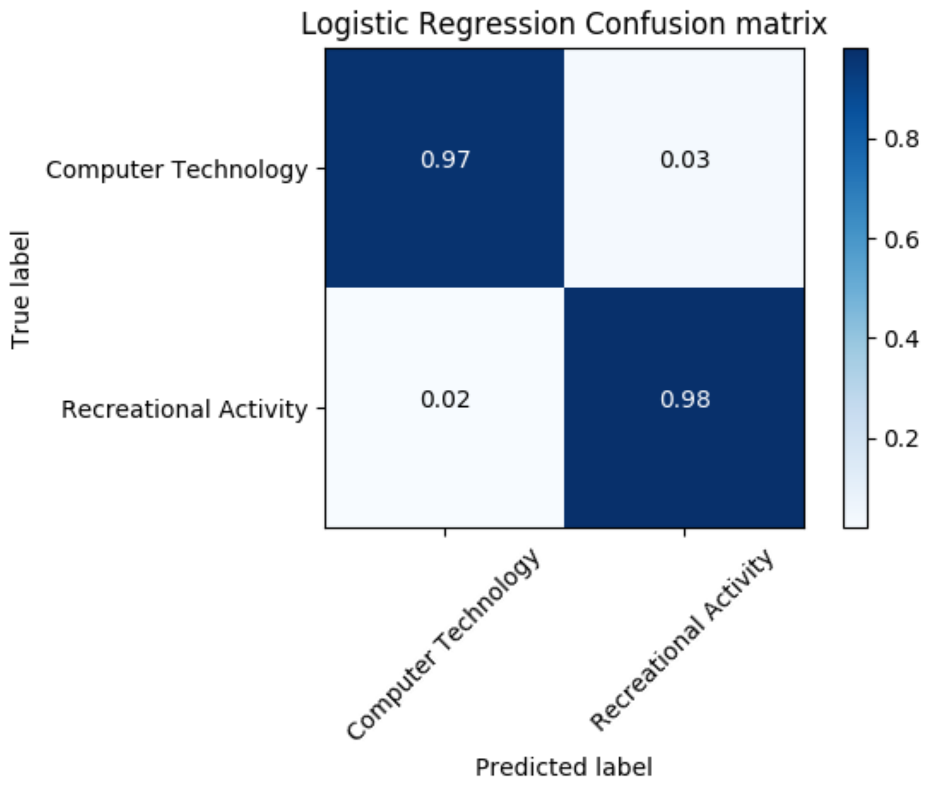

Figure 9: Confusion Matrix for logistic classifier with no regularization

We then trained logistic classifiers using L1 and L2 regularization. We used 5-fold cross validation to search for the best C parameter in the range $\{10^k | -3 \leq k \leq 3, k \in \mathbb{Z}\}$. The results of the cross validation are shown below. We found that the optimal C was 10 for L1 regularization and 100 for L2 regularization.

After using cross-validation to find the best C, we re-trained the classifiers with the optimal C values, and compared these with the classifier using no regularization on the test set. The performance of all 3 classifiers are shown in the table below. We found that the classifier with L2 regularization performs the best across all metrics.

To see the effect of the regularization parameter C on the test error, we plotted the test accuracy over several values of C shown below. When $C = 0.001$, we have a large regularization strength, which can cause the model to underfit and have a poor test accuracy. As C increases from 0.001, the accuracy also increases at first. However, this accuracy levels off as C gets too large.

We found that small values of C force the coefficients of the logistic classifier to be smaller, while large values of C allow larger coefficients. We calculated the average

C	L1 Regularization	L2 Regularization
0.001	0.49513943	0.68470286
0.01	0.89011070	0.94843623
0.1	0.96322847	0.96914546
1	0.97083724	0.97252723
10	**0.97612020**	0.97569624
100	0.97527520	**0.97654258**
1000	0.97506401	0.97548662

Table 1: Cross Validation accuracy averaged over 5 folds.

Regularization	Accuracy	Precision	Recall	F1-score
None	0.97365079	0.96772191	0.98050314	0.97407060
L1	0.97238095	0.96532508	0.98050314	0.97285491
L2	**0.97428571**	**0.96776193**	**0.98176101**	**0.97471121**

Table 2: Performance metrics for logistic classifiers with L1, L2, and no regularization on the test set.

coefficient magnitude for each C, shown in the graph below.
We also found that L1 regularization performs selection on the coefficients by forcing some of them to be 0. For small values of C we observed that most of the coefficients were 0 in L1 regularization. L2 regularization does not perform selection on the coefficients, but rather shrinks their magnitude. On average, the coefficients found in our models with L2 regularization were smaller than the coefficients would with L1 regularization. The fact that L1 regularization performs selection on the data features makes L1 regularization a good choice when dealing with a large number of features. However, L1 regularization does not have a closed form solution, making calculations slower than with L2 regularization. Thus L2 regularization would be more ideal in situations with a large number of training samples.

Question 6

Following the approach of the previous two classifiers, we proceeded to apply a *GaussianNB* classifier on the training data as only a functional classifier. The training data was first cleaned using TF-IDF and LSI methods using the code below:

```python
if __name__ == "__main__":
    # Using similar framework as Q5
    X_train_tfidf, X_test_tfidf, target_Train, target_Test = getData()
    y1, y2, _ = LSI_(X_train_tfidf, X_test_tfidf)
```

The data was now passed into the Naive Bayes classifier and the type of classifier was set to a Gaussian. The code below shows the function definition and the fit and

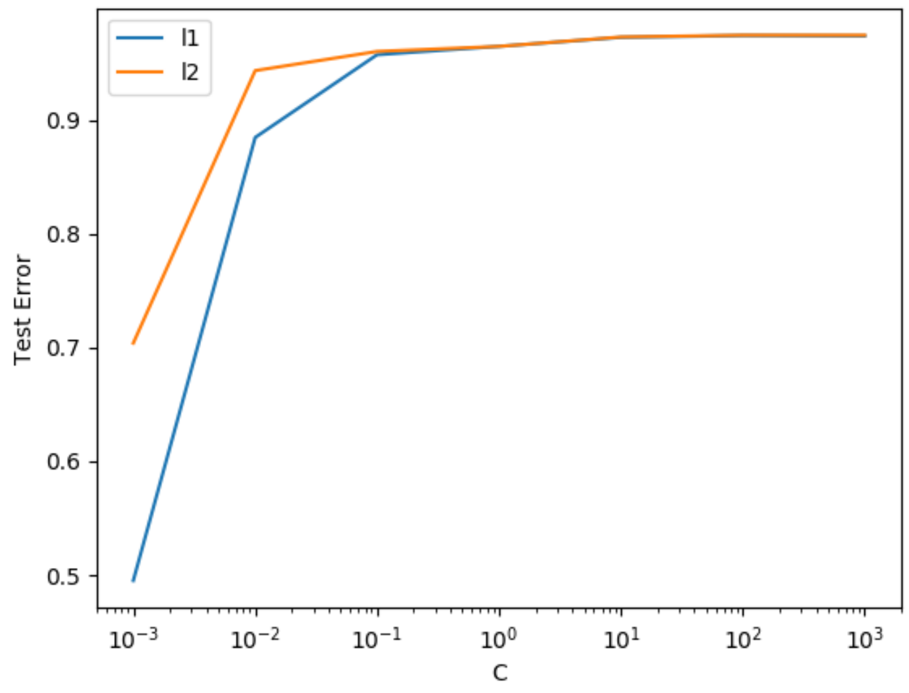

Figure 10: Test Accuracy over various values of C

predict methods called to train and test the model.

```
def Naive_Bae(X_train_r, y_train, X_test_r, y_test):
    clf = GaussianNB().fit(X_train_r, y_train)
    fit_predict_and_plot_roc1(clf, X_test_r, y_test)
    cnf_matrix = confusion_matrix(y_test, clf.predict(X_test_r))
    plt.figure()
    class_names = ['Computer Technology', 'Recreational Activity']
    plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=
   True, title='Bayes Classifier Confusion matrix')
    plt.show()
    print_metrics(y_test, clf.predict(X_test_r))
```

We then printed the parameters accuracy, precision, recall and F1-score as shown below. The values indicate that the classifier is performing at roughly 90% accuracy with the other parameters performing equally as well:

```
('X_train_tfidf size: ', (4732, 16292))
('X_test_tfidf  size: ', (3150, 16292))
Naive Bayesian Classifier:
Normalized confusion matrix
[[0.85512821 0.14487179]
 [0.04025157 0.95974843]]
('Accuracy: ', 0.9079365079365079)
```

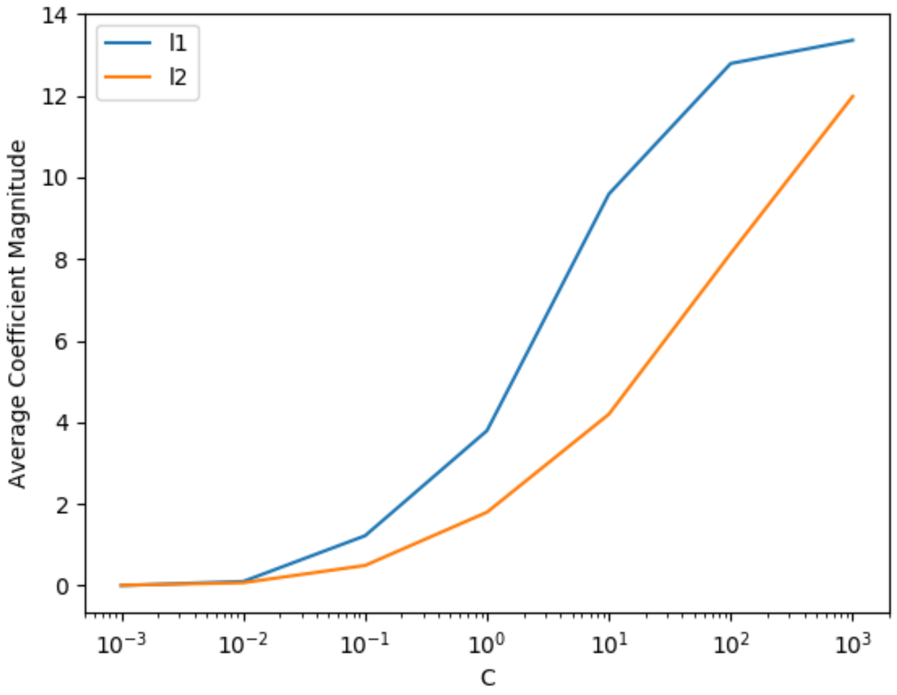

Figure 11: Average coefficient magnitudes over various values of C

```
('Precision: ', 0.8710045662100456)
('Recall: ', 0.959748427672956)
('F1-score: ', 0.9132256134051466)
```

The ROC curve and Confusion matrix plotted confirmed the expectation that the model was performing as expected. The figures are shown below:

ROC Curve for a GaussianNB classifier

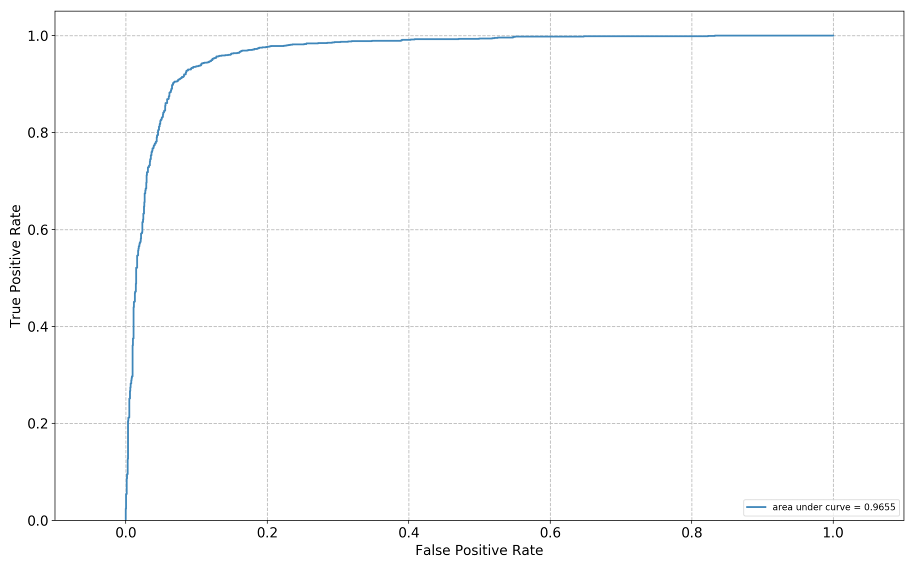

Figure 12: The Naive Bayes ROC curve covers enough area so as to be a reasonable model for binary classification

Confusion Matrix for a GaussianNB classifier

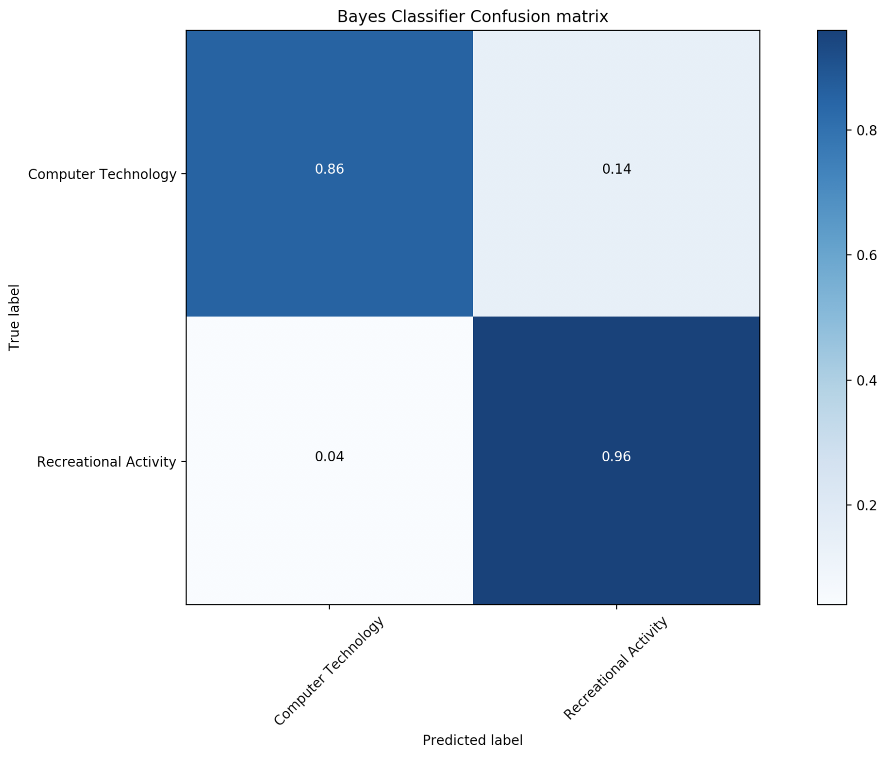

Figure 13: The ROC confusion matrix is major diagonal heavy implying a well-performing classifier

Question 7

In the previous sections, we successfully trained and tested different classifiers. Now, we attempt to find the best classifier with the most optimal hyper-parameters. To do this, we performed a grid search of all the different combinations of options shown in the table below.

Procedure	**Options**
Loading Data	remove "headers" and "footers" vs not
Feature Extraction	`min_df = 3` vs 5; use lemmatization vs not
Dimensionality Reduction	LSI vs NMF
Classifier	SVM with the best γ previously found vs Logistic Regression: L1 regularization vs L2 regularization, with the best regularization strength previously found vs `GaussianNB`
Other options	Use default

We executed this by constructing a *Pipeline* in python to perform the same feature extraction, dimensionality reduction and classification as was performed in previous problems:

```
pipeline = Pipeline([
    ('vectorizer', CountVectorizer(min_df=1, stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('reduce_dim', TruncatedSVD(random_state=0)),
    ('classifier', GaussianNB()),
],
memory=memory
)
param_grid = [{
    'vectorizer__min_df': MIN_DF_OPTIONS,
    'vectorizer__analyzer': ANALYZER_OPTIONS,
    'reduce_dim': REDUCE_DIM_POSS,
    'classifier': [LinearSVC(C=gamma_10)]
              },{
    'vectorizer__min_df': MIN_DF_OPTIONS,
    'vectorizer__analyzer': ANALYZER_OPTIONS,
    'reduce_dim': REDUCE_DIM_POSS,
    'classifier': [LogisticRegression(C=c_10, solver='liblinear')],
    'classifier__penalty':l1
```

```
              },{
    'vectorizer__min_df': MIN_DF_OPTIONS,
    'vectorizer__analyzer': ANALYZER_OPTIONS,
    'reduce_dim': REDUCE_DIM_POSS,
    'classifier': [LogisticRegression(C=c_100, solver='liblinear')],
    'classifier__penalty':l2
              },{
    'vectorizer__min_df': MIN_DF_OPTIONS,
    'vectorizer__analyzer': ANALYZER_OPTIONS,
    'reduce_dim': REDUCE_DIM_POSS,
    'classifier': [GaussianNB()]
}]
```

Where:

```
# Pipeline options
MIN_DF_OPTIONS = [3,5]
ANALYZER_OPTIONS = ['word', stem_rmv_punc] #with and without lemm
REDUCE_DIM_POSS = [TruncatedSVD(n_components=50, random_state=0), NMF(
    n_components=50, init='random', random_state=0)]
l1 = ['l1']
l2 = ['l2']
c_10 = 10
c_100 = 100
gamma_10 = 10
```

We ended up with two dataframes, corresponding to our results with headers and footers removed from the data, and headers and footer not removed from the data. From these dataframes, we look at the given mean_test_score to determine the best combination of parameters.

In our case, we see that we got our best results when:

- Headers and footers included
- min_df = 3
- Use lemmatization
- LSI for dimensionality reduction
- Logistic Regression Classifier with l1 regularization (C = 10)
- Remaining Options Defaulted

In our results the SVM and logistic regression classifiers performed very similarly, and the naive Bayes classifier performed the worst. We also observed that the classifiers using LSI for dimension reduction performed better than the ones using NMF in almost all cases. The exception is the Bayes classifier, in which NMF gave better results. Lastly, we found that there was very little difference in performance when using `min_df=3` vs `min_df=5` and when removing headers and footers vs including them.

Question 8

Having exhausted possible options in binary classification, this section was dedicated to multiclass methods: specifically, the classes 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'misc.forsale', 'soc.religion.christian'.
The data was obtained using the following code segment:

```
class_names = ["comp.sys.ibm.pc.hardware", "comp.sys.mac.hardware","misc
    .forsale", "soc.religion.christian"]

X_train_tfidf, X_test_tfidf, Target_Train, Target_Test = vectorize_data(
    class_names, binary = False)

X_train_r, X_test_r, _ = LSI_(X_train_tfidf, X_test_tfidf)
```

The multi-class SVM (for both versions, one v. one and one v. all) used a grid search to find the ideal hyper-parameter for C in the training data over 5 cross-validated instances. This hyper-parameter was applied globally to all sub-classifiers in the composite model. The classifiers followed similar approaches in training and testing; the code snippet for the *OneVsRestClassifier* is provided below: (opt is set to 1 in main)

```
parameters = {'estimator__C':[0.001, 0.01, 0.1, 1, 10, 100, 1000]}
if opt == 1:
        ovr = OneVsRestClassifier(LinearSVC(class_weight = "balanced"))
        clf = GridSearchCV(ovr, parameters, cv=5).fit(X_train_r,
    Target_Train)
        title = 'One v All Confusion matrix'

cnf_matrix = confusion_matrix(Target_Test, clf.predict(X_test_r))
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,  normalize=True,
    title=title)
plt.show()
print_metrics(Target_Test, clf.predict(X_test_r), "macro")
```

Note that we set class weights to be balanced; in a one vs. rest classifier with equally sampled classes, a binary SVM with a one v. rest approach would be severely unbalanced by a factor of roughly 3. Running this super-classifier with the 4 balanced SVM sub-classifiers, the following confusion matrix was formed along with its measures:

```
('Accuracy: ', 0.8849840255591054)
('Precision: ', 0.8847579159206288)
('Recall: ', 0.8844453225644899)
('F1-score: ', 0.8844729711123398)
```

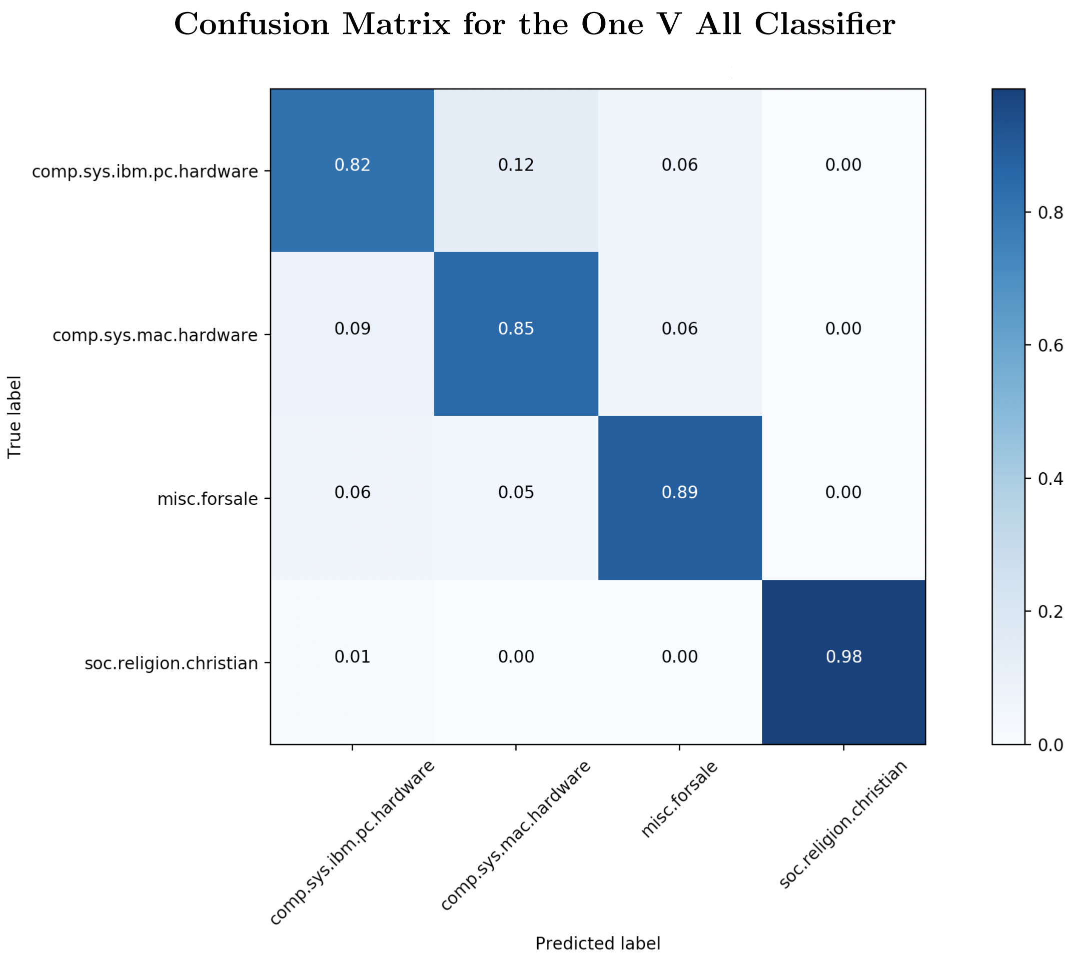

Figure 14: The One v All Confusion Matrix: (Note the 4 features instead of only two)

Similarly, we created a *OneVsOneClassifier* super-classifier that grid-searched for the ideal hyper parameter to apply to the 6 SVM sub-classifiers. The balanced option was not required here, as the classifier would only classify per pair of groups (these were found to already have balanced samples). Once again, the measures and the confusion matrix are given below:

```
('Accuracy: ', 0.870926517571885)
('Precision: ', 0.8712650422711707)
('Recall: ', 0.8703125347373374)
('F1-score: ', 0.8707454046020168)
```

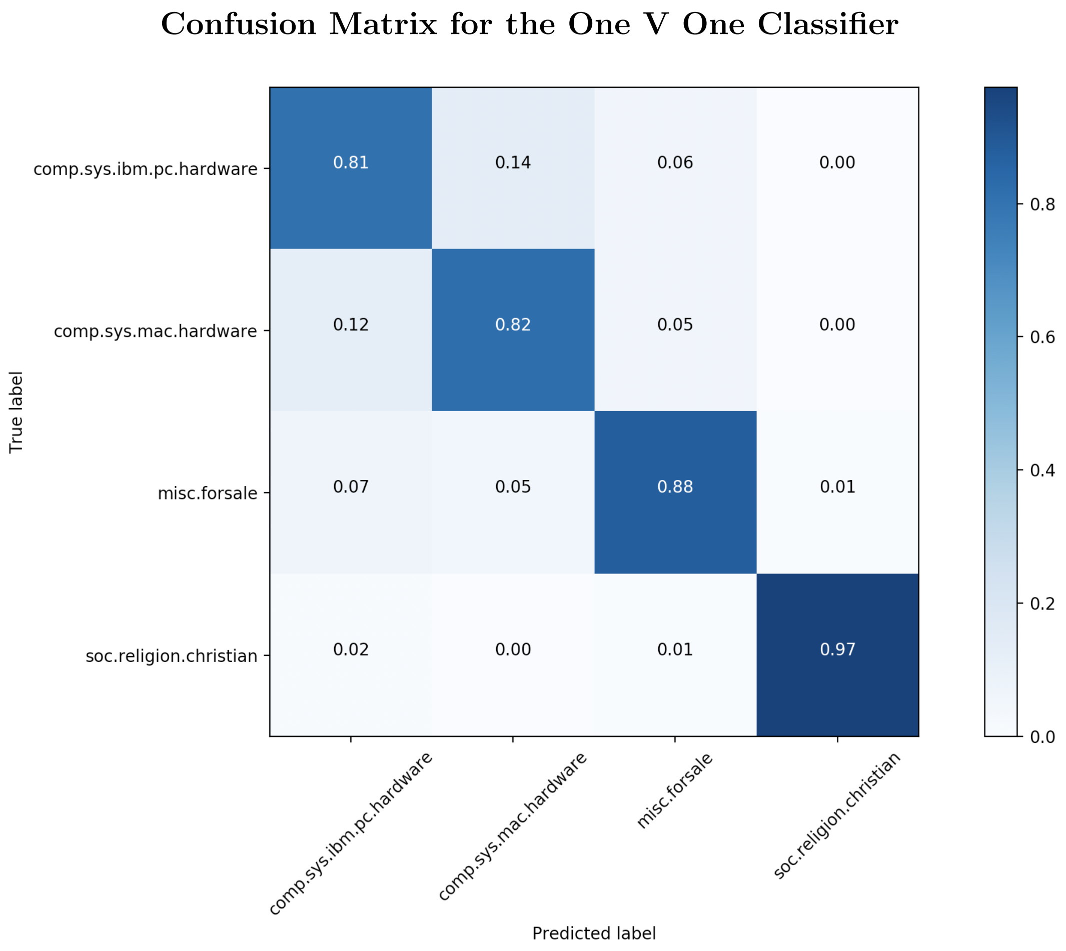

Figure 15: The One v One Matrix: (Note the 4 features instead of only two)

Lastly, we extend the multiclass classification to the Naive Bayes method where we use the *GaussianNB* classifier, which naturally classifies multiple classes. The measures and confusion matrix are given below. Note that it performs poorest.

```
('Accuracy: ', 0.6952076677316293)
('Precision: ', 0.7056475107074915)
('Recall: ', 0.6929827246813967)
('F1-score: ', 0.6874863799734776)
```

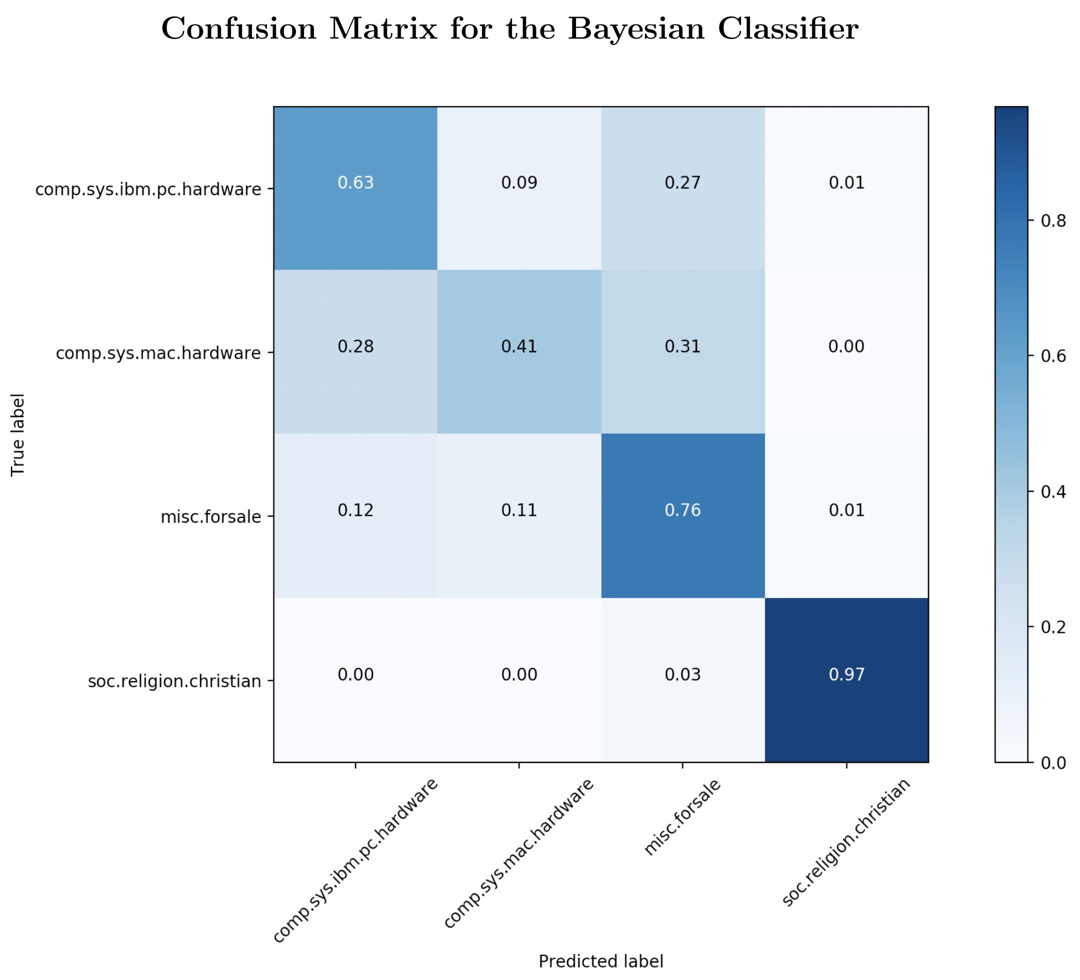

Figure 16: The Gaussian NB Classifier Matrix: (Note the 4 features instead of only two)

Conclusion

We have thus verified that the Bag of Words model when applied with smart feature selection and dimensionality reduction performs reasonably in both binary classification and multiclass classification, and is a great model for context classification of textual data.