EE219 - UCLA

WINTER 2019

PROJECT 1 : REPORT

Classification Analysis on Textual Data

GROUP MEMBERS

Anchal Goyanka

Nandan Parikh

Pratik Mangalore

Srishti Majumdar

INTRODUCTION

The project deals with statistical classification of textual data from a well defined “20 Newsgroup dataset”. The dataset deals with 20,000 documents partitioned into 20 different newsgroups like computer hardware, sport sections, religion and so on.

There are different sections to the project which can be broadly divided as follows :

1. Exploring the dataset to get familiar with the various newsgroup and the data distribution
2. Extracting features from the documents - bag of words model
3. Applying transformations on the terms by using stemming, lemmatization along with removal of stop words and punctuations
4. Dimensionality reduction of the model as there are thousands of dimensions in the initial model. Using PCA and NMF for the same
5. Applying classification algorithms on this dataset by fine tuning the parameters for each algorithm. Plotting the ROC curves, checking the F-1 score and the confusion matrix to see how each algorithm behaves
6. Creating pipelines for the algorithms to make the process easier and then performing grid search.

Question 1: Analyzing the data

We are dealing with the classification problem using the 20 Newsgroup dataset. Before we start applying transformations on the data we need to check if the data is balanced across various classes.

To do this, we check the number of documents relating to each class/newsgroup. By plotting the histogram for each newsgroup we can visualize the various sizes present.

The histogram plot looks as below for all categories in the dataset :

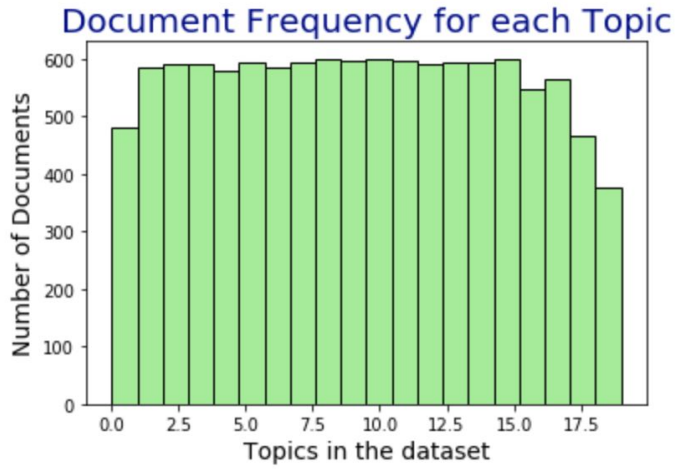


Figure 1.1: Histogram for document frequency of all categories

If we only consider the subset of the 8 categories we will be working the histogram looks as follows. It is clear that the data is divided equally.

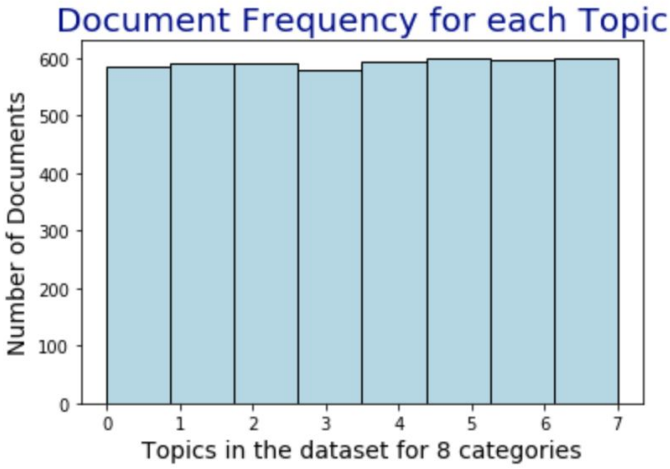


Figure 1.2: Histogram for document frequency of only 8 categories

Question 2 : Feature Extraction

In this section we are only dealing with 8 categories :

```
comp_categories = [ 'comp.graphics', 'comp.os.ms-windows.misc',
                    'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware']
rec_categories = ['rec.autos', 'rec.motorcycles',
                  'rec.sport.baseball', 'rec.sport.hockey']
```

Figure 2.1: Categories used

From these categories the primary step is proper representation of documents. The representation method used for this project is the ‘Bag of Words’ where the frequency of each term in a document is represented in a matrix form. Here we get a matrix of *n* documents and *m* terms.

We see that the number of the terms is quite large and many of these terms may not be required. As every distinct term has a frequency, we need to remove some of the unnecessary words. We do this by removing all the ‘stopwords’ in the vocabulary as these words add no value to our representation. Next we remove the punctuations and the digits from the corpus of words as these do not add value.

We also want to map the different forms of the same word to the same root word, e.g. "talks", "talking", "talked" should all be the same as "talk". We use two specific techniques for this :

1. Stemming: Trying to shorten a word with simple regex rules
2. Lemmatization: Trying to find the root word with linguistics rules

Adding this all up, we use countVectorizer with stopwords, removing digits and applying stemming and lemmatization to create our first representation of the dataset. Here min_df points to the terms that have a document frequency strictly lower than the given threshold which is 3 in this case.

```
#2 : Main function starts here
Using the countVectorizer with min_df as 3 and stopwords along with the custom analyzer
X_train_counts1 contains the result of applying countVectorizer
X_test_counts1 transforms according to the train data
'''

count_vect1 = CountVectorizer(min_df=3,analyzer=stem_rmv_punc, stop_words='english')
X_train_counts1 = count_vect1.fit_transform(twenty_train.data)
X_test_counts1 = count_vect1.transform(twenty_test.data)

print(X_train_counts1.shape)
print(X_test_counts1.shape)
```

```
(4732, 16292)
(3150, 16292)
```

Figure 2.2: Countvectorizer with required parameters

The next step is performing the tf-idf on this representation. In a way we are normalizing the data with a certain function of the frequency of individual words. Intuitively, this term weighing system will assign the largest weight to those terms which with high frequency in individual documents but are at the same time relatively rare in the collection as a whole.

```
#2 : Using the tfidf transformer on the modified data and checking the shape of the result
'''

from sklearn.feature_extraction.text import TfidfTransformer
tfidf_transformer = TfidfTransformer()

X_train_result = tfidf_transformer.fit_transform(X_train_counts1)
X_test_result = tfidf_transformer.transform(X_test_counts1)

print(X_train_result.shape)
print(X_test_result.shape)
```

```
(4732, 16292)
(3150, 16292)
```

Figure 2.3: TFIDF transformer on the transformed data from Figure 2.2

We see the shape of the matrix to be **4732 x 16292** for the train data and **3150 x 16292** for the test data.

Question 3: Dimensionality Reduction

The major problem we can infer from the shape is the huge dimension of terms. We have around 16k terms and if we try to classify our data based on these parameters it can take an incredible amount of time. Our algorithms may perform poorly in such a scenario, which is also referred to as the 'Curse of Dimensionality', and would lower the statistical significance of our data.

The goal is to select a subset of these features which are more relevant for classifying the data by reducing the dimensions. The two approaches which were discussed in the class have been used for this : Latent Semantic Indexing (LSI) and Non-negative Matrix Factorization (NMF).

We want to reduce our model to a 50 dimension matrix. We performed LSI and NMF with k = 50 to get the results.

```
'''
#3 - Dimensionality reduction
Non Negative Matrix Factorization using NMF library
Reduce dimensions to 50 for the 8 category dataset
'''

from sklearn.decomposition import NMF

model = NMF(n_components=50, init='random', random_state=None)
W_train = model.fit_transform(X_train_result)
W_test = model.transform(X_train_result)

print(W_train.shape)
print(W_test.shape)
```

```
(4732, 50)
```

```
'''
#3 - Dimensionality reduction
LSI Latent Semantic Indexing using TruncatedSVD library
Reduce dimensions to 50 for the 8 category dataset
'''

from sklearn.decomposition import TruncatedSVD

svd = TruncatedSVD(n_components=50, random_state=None)
X_train_reduced = svd.fit_transform(X_train_result)
X_test_reduced = svd.transform(X_test_result)

print(X_train_reduced.shape)
print(X_test_reduced.shape)
```

```
(4732, 50)
(3150, 50)
```

Figure 3.1: Code for dimensionality reduction using LSI and NMF

Next, after reducing the dimensions we check which method performed better. In other words, which model had less loss of data during the process.
For both the model we compute the loss and see that ***LSI performed better*** as it had less loss.

Reasoning for the better performance of LSI(SVD) as compared to NMF according to us is as follows :

In SVD the importance of each vector in the basis is relative to the singular value of that vector. This means that the first vector usually dominates and is the most used vector to reconstruct the data. This goes on for the 50 dimensions that are created. Due to this implicit hierarchy in SVD which is not present in NMF, SVD has less loss as we have the most important basis sorted. NMF on the other hand, reconstructs each vector as a positive sum of basis vectors, which means taking a little from each basis which may lead to an optimal solution, but not as good as SVD.

The approximate loss for LSI was 4108 and that for NMF was 4150.

```
'''
For the LSI model - checking the loss
'''

V = svd.components_
print(V.shape)

print("The difference for LSI model is : ")
LSIDiff = np.sum(np.array(X_train_result - X_train_reduced.dot(V))**2)
print(LSIDiff)
```

```
(50, 16292)
The difference for LSI model is :
4108.609474988852
```

```
'''
For the NMF model - checking the loss : || X - WH ||^2
'''
H = model.components_
print(H.shape)
print("The difference for NMF model is : ")
NMFDiff = np.sum(np.array(X_train_result - W_train.dot(H))**2)
print(NMFDiff)
```

```
(50, 16292)
The difference for NMF model is :
4150.704627414409
```

Figure 3.2: Code for finding the loss during dimensionality reduction

Question 4: Classification using SVMs

Starting this point, we have reduced those 8 categories from Figure 2.1 to two categories, “Computer Technology” and “Recreational Activity” which are represented by values 0 and 1 respectively.

```
def combineDocs(twenty):
  del twenty.target_names[:]
  match =[0,1,2,3]
  for i in range(len(twenty.data)) :
    if twenty.target[i] in match :
      twenty.target[i] = 0
    else :
      twenty.target[i] = 1

  twenty.target_names = ['Computer Technology', 'Recreational Activity']
  return twenty
```

Figure 4.1 Code to combine categories

The implementation of Linear SVM using LinearSVC model from scikit-learn svm library can be seen in Figure 4.1.

In [46]:

```
#using linearsvc
''' Question 4:
Train two linear SVMs and compare:
- Train one SVM with γ = 1000 (hard margin), another with γ = 0.0001 (soft margin).
- Plot the ROC curve, report the confusion matrix and calculate the accuracy, recall,
precision and F-1 score of both SVM classifier. Which one performs better?
- What happens for the soft margin SVM? Why is the case?
* Does the ROC curve of the soft margin SVM look good? Does this conflict with
other metrics?
'''

from sklearn.svm import LinearSVC
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
# first we will train the linear svm with gamma= 1000 and gamma= 0.0001
svm_1000= LinearSVC(C=1000)
svm_1000.fit(X_train_reduced,twenty_train.target)
print("Done with SVM with Margin 1000")

svm_0001= LinearSVC(C=0.0001)
svm_0001.fit(X_train_reduced, twenty_train.target)
print("Done with SVM with Margin 0.0001")

# now we check how well they can predict the test set
y_pred_1000= svm_1000.predict(X_test_reduced)
y_pred_0001= svm_0001.predict(X_test_reduced)
```

Figure 4.2: Implementation using Linear SVC

The functions used to calculate and display the metrics required can be seen in Figure 4.2.

```
#checking the confusion matrix, this will give us recall, precision, f1 score along with the support for these scores
print("Scoring details for gamma= 1000")
print("Confusion Matrix")
print(confusion_matrix(twenty_test.target,y_pred_1000))
print("Report")
print(classification_report(twenty_test.target,y_pred_1000))
print("Accuracy")
print(accuracy_score(twenty_test.target,y_pred_1000))
print("ROC Plot")
roc_plotting("SVM (gamma=1000)", y_pred_1000, twenty_test.target)
print("Scoring details for gamma=0.0001")
print("Confusion Matrix")
print(confusion_matrix(twenty_test.target,y_pred_0001))
print("Report")
print(classification_report(twenty_test.target,y_pred_0001))
print("Accuracy")
print(accuracy_score(twenty_test.target,y_pred_0001))
print("ROC Plot")
roc_plotting("SVM (gamma= 0.0001)", y_pred_0001, twenty_test.target)
```

Figure 4.3: Implementation to Retrieve Scores

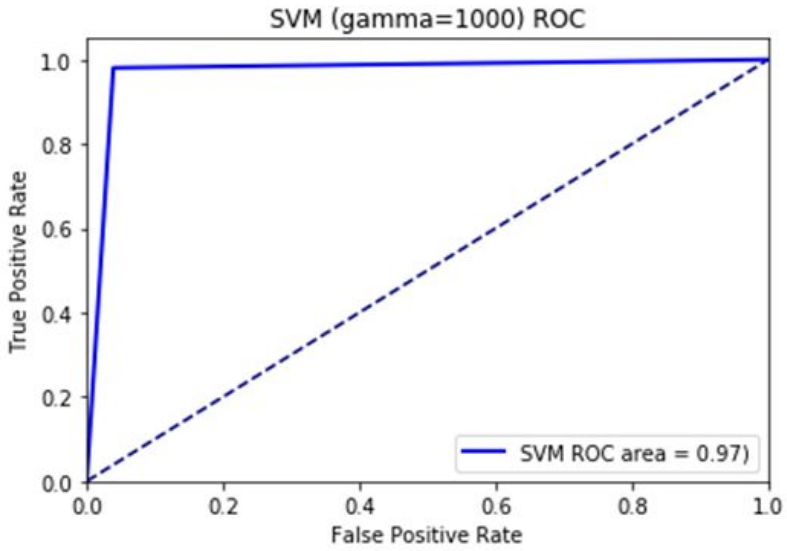
Metrics obtained for the SVM with hard margin γ = 1000 is as follows:

Confusion Matrix		Predicted Labels	
		0	**1**
True Labels	**0**	1499	61
	1	21	1559

Classification Report	Precision	Recall	F1-Score	Support
0	0.98	0.96	0.97	1560
1	0.96	0.98	0.97	1590
Micro avg	0.97	0.97	0.97	3150
Macro avg	0.97	0.97	0.97	3150
Weighted avg	0.97	0.97	0.97	3150

Accuracy: 0.9707936507936508

ROC Plot:



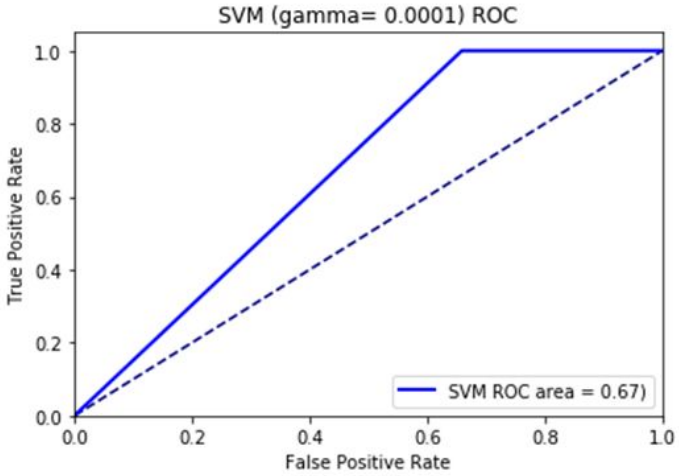
Metrics obtained for the SVM with hard margin γ = .0001 is as follows:

Confusion Matrix		Predicted Labels	
		0	**1**
True Labels	**0**	532	1028
	1	0	1590

Classification Report	Precision	Recall	F1-Score	Support
0	1.00	0.34	0.51	1560
1	0.61	1.00	0.76	1590
Micro avg	0.67	0.67	0.67	3150
Macro avg	0.80	0.67	0.63	3150
Weighted avg	0.80	0.67	0.63	3150

Accuracy: 0.6736507936507936

ROC Plot:



The implementation of linear SVM using SVC model with linear kernel from scikit-learn svm library is similar to the implementation shown in Figure 4.2. The retrieval of scores remains as per Figure 4.3.

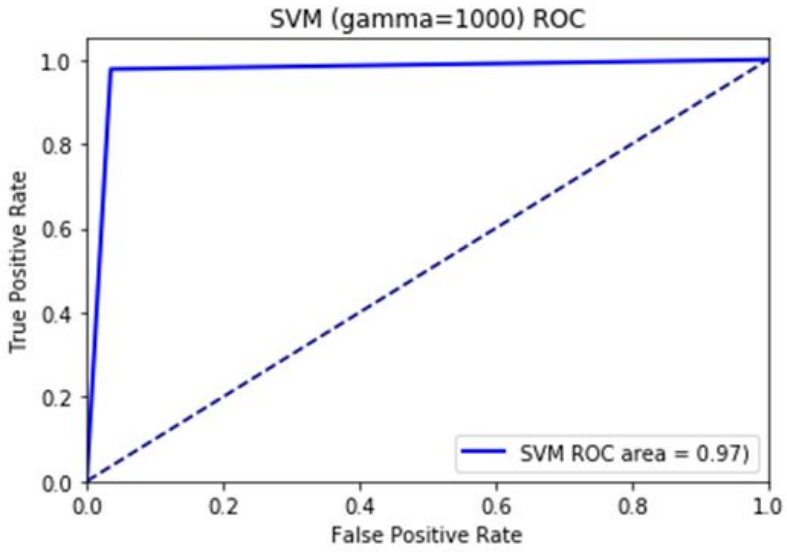
Metrics obtained for the SVM (using SVC) with hard margin γ = 1000 is as follows:

Confusion Matrix		Predicted Labels	
		0	**1**
True Labels	**0**	1505	55
	1	36	1554

Classification Report	Precision	Recall	F1-Score	Support
0	0.98	0.96	0.97	1560
1	0.97	0.98	0.97	1590
Micro avg	0.97	0.97	0.97	3150
Macro avg	0.97	0.97	0.97	3150
Weighted avg	0.97	0.97	0.97	3150

Accuracy: 0.9711111111111111

ROC Plot:



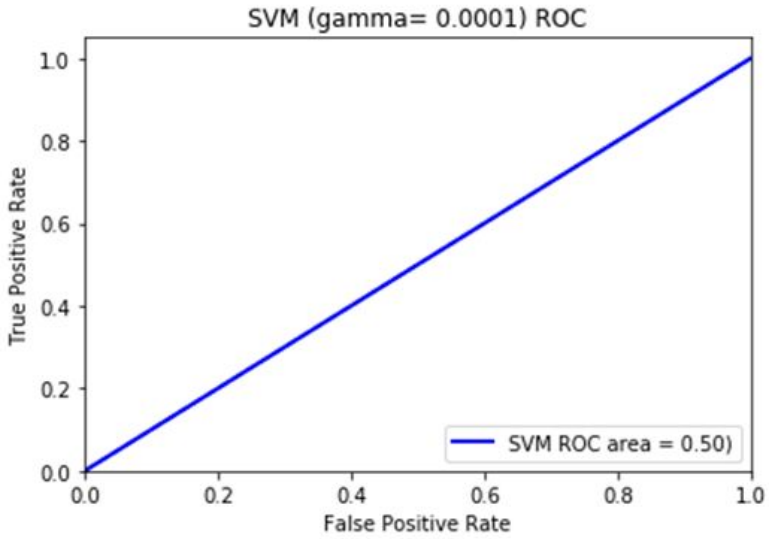
Metrics obtained for the SVM with hard margin γ = .0001 is as follows:

Confusion Matrix		Predicted Labels	
		0	**1**
True Labels	**0**	0	1560
	1	0	1590

Classification Report	Precision	Recall	F1-Score	Support
0	0.00	0.00	0.00	1560
1	0.50	1.00	0.67	1590
Micro avg	0.50	0.50	0.50	3150
Macro avg	0.25	0.50	0.34	3150
Weighted avg	0.25	0.50	0.34	3150

Accuracy: 0.5047619047619047

ROC Plot:



As we can see from the performance scores obtained, the linear SVM with hard margin, that is, γ = 1000, performs better than the linear SVM with soft margin, γ = .0001 .

The γ value is denoted by parameter 'C' in LinearSVM and SVC. This parameter decides how well our classifier will avoid misclassifications. Hence for smaller values for C, we have a higher tendency to get misclassified examples compared to higher values of C even when are data is linearly separable.

The ROC curve for the classifier with soft margin depicts that the accuracy of the classifier is low and performs relatively poorly. Looking at the confusion matrix we see that there are no False Positives and low True Positives using LinearSVC as well as no False Positives and no True Positives using SVC. This shows that classifier seems to assign most data points to one label at C=0.0001.

For the 5-fold Cross validation, the code implemented is shown in Figure 4.4.

```
In [17]:
'''CV with grid search
'''

from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.model_selection import ShuffleSplit

parameters={'C': [10**k for k in range(-3,4)]}
gsc= GridSearchCV(LinearSVC(), parameters, scoring='accuracy', n_jobs=-1, cv=5, return_train_score=True)
#gsc= GridSearchCV(SVC(kernel='linear'), parameters, scoring='accuracy', n_jobs=-1, cv=5, return_train_score=True)
gsc.fit(X_train_reduced, twenty_train.target)
print("Best Parameters: %s" % gsc.best_params_)
```

Figure 4.4: Searching for Best Parameters

The results using LinearSVC and SVC with linear kernel are almost exactly same for all. They are as follows:

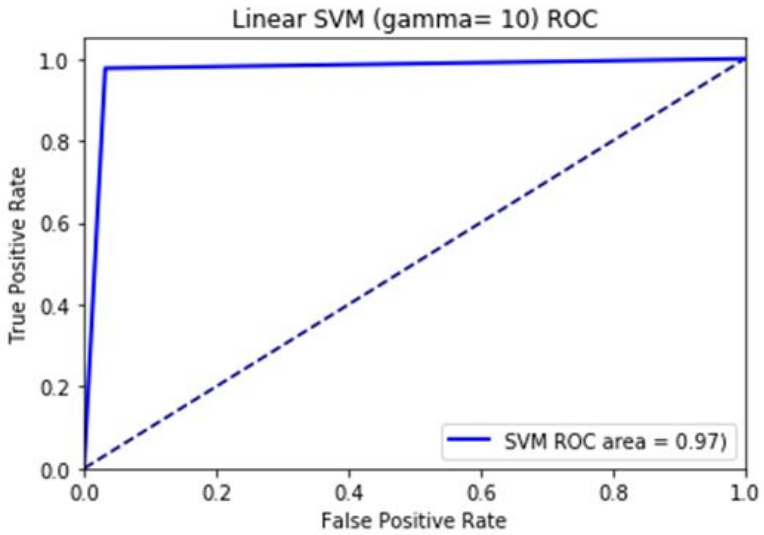
Best Parameter: γ=10

Confusion Matrix		Predicted Labels	
		0	1
True Labels	0	1510	50
	1	37	1553

Classification Report	Precision	Recall	F1-Score	Support
0	0.98	0.97	0.97	1560
1	0.97	0.98	0.97	1590
Micro avg	0.97	0.97	0.97	3150
Macro avg	0.97	0.97	0.97	3150
Weighted avg	0.97	0.97	0.97	3150

Accuracy: 0.9723809523809523

ROC Plot:



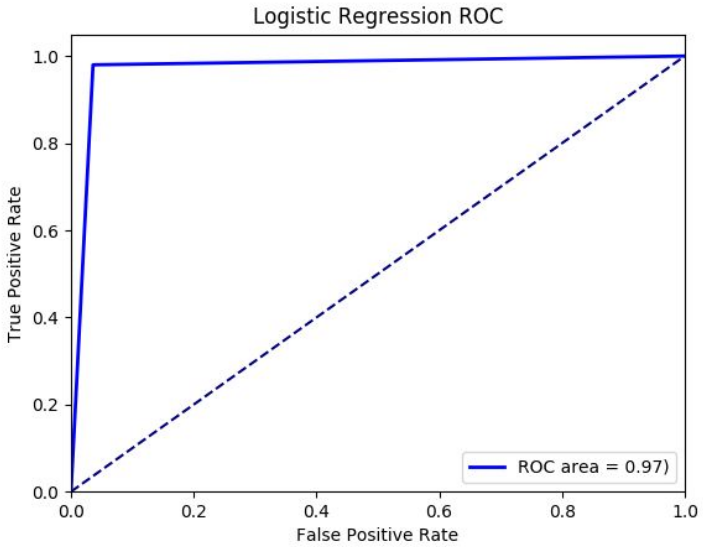
Question 5: Logistic Regression

In order to train a logistic classifier without regularization, we use the LogisticRegression model from sklearn and set the regularization constant c = 10^10. In the loss function, the regularization term is multiplied by inverse of c and thus c being a large number this model behaves as if there was no regularization. Using L2 norm with it, we a get the following values:

Confusion Matrix		Predicted Labels	
		0	**1**
True Labels	**0**	1505	55
	1	37	1553

Classification Report	Precision	Recall	F1-Score	Accuracy
Weighted avg	0.97	0.97	0.97	0.9704

ROC Plot:



```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.linear_model import LogisticRegression
LR = LogisticRegression(solver='liblinear',C=10000000000, penalty="l2")
clf = LR.fit(X_train_reduced, twenty_train.target)
y_pred = clf.predict(X_test_reduced)
print("Confusion Matrix")
print(confusion_matrix(twenty_test.target,y_pred))
print("Report")
print(classification_report(twenty_test.target,y_pred))
print("Accuracy")
print(accuracy_score(twenty_test.target,y_pred))
```

Figure 5.1 Code implementing Logistic Regression

Regularization:

After using 5-fold cross-validation on the training data, the best regularization strength for L1 regularization is 10 and for L2 regularization is 100.

In order to compare the performance of the 3 kinds of logistic classifiers, let us look at their Individual scores as below:

	w/o regularization	L1 regularization	L2 regularization
Precision	0.97	0.97	0.97
Recall	0.97	0.97	0.97
Accuracy	0.9707	0.9720	0.9720
F1-score	0.97	0.97	0.97

Looking at the scores for the three models above, we see that precision, recall, accuracy and F1-score are the same across all the regularization models for the given dataset.

The test error and learning coefficients are all same across different regularization parameters for the different dataset. But one can look at regularization from Bayesian point of view and might be interested in doing logistic regression without regularization when one wants to give weightage to only data and not imbibe a prior knowledge in the model. Similarly, one can use L1 and L2 regularizations when one wants to put some prior knowledge into the model and not completely trust the data, as the dataset could be noisy.

L1 regularization is more robust, but can provide multiple solutions whereas L2 regularization provides stable single solutions. L2 is also more sensitive to error, and hence the model fits noise even lesser than L1 regularization.

Both logistic regression and linear SVM are trying to classify data points using a linear decision boundary but they have completely different ways to find this boundary. In SVM, one tries to maximize the margin between support vectors and optimize for it. While in Logistic Regression one maximizes the posterior and used MAP measure.
The results from SVMs are deterministic while the results from Logistic Regression is probabilistic. Even though it is a convex optimization problem, methods such as gradient descent and Newtonian's methods are used to evaluate the answers. This might affect their performances in many cases. Also one can use kernels with SVMs and make the data linearly separable in higher dimensions. This makes it faster as well as scalable which might affect its performance. Also, SVMs usually work much better in higher dimensions than logistic regression.

Question 6: Naive Bayes

The gaussian naive bayes classifier, models the likelihood of the data using the normal distribution. We have used the GaussianNB model from sklearn as follows

In [19]:
```
'''
Question 6:

Naïve Bayes classifier: train a GaussianNB classifier; plot the ROC curve and
report the confusion matrix and calculate the accuracy, recall, precision and F-1 score of this
classifier.
'''

from sklearn.naive_bayes import GaussianNB
clf_NB = GaussianNB().fit(X_train_reduced, twenty_train.target)
test_labels = clf_NB.predict(X_test_reduced)
accuracy_list = test_labels==twenty_test.target
accuracy=sum(accuracy_list)/3150
print("Confusion Matrix")
print(confusion_matrix(twenty_test.target,test_labels))
print("Report")
print(classification_report(twenty_test.target,test_labels))
print("Accuracy")
print(accuracy_score(twenty_test.target,test_labels))
print("ROC Plot")
roc_plotting("Gaussian Naive Baise", y_pred_gamma, twenty_test.target)
```

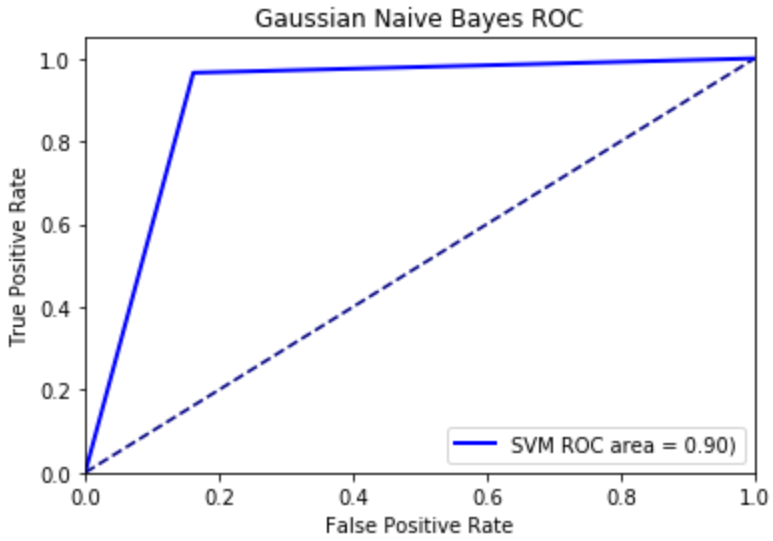
Here are the observations of performing the classification -

Confusion Matrix		**Predicted Labels**	
		0	**1**
True Labels	**0**	1347	213
	1	86	1504

Classification Report	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.96	0.84	0.89	1560
1	0.86	0.97	0.91	1590
Weighted avg	0.91	0.90	0.90	3150

Accuracy: **0.9025396825396825**

ROC Plot:



Question 7 : Grid Search of Parameters

We need to create pipeline(s) that performs feature extraction, dimensionality reduction and classification.
The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. It saves up a lot of time as well as makes working on the whole process intuitive. First we used the pipelines to check the results we got previously. We created three separate pipelines first as below to understand the working of the whole model.

The pipelines used (similar to one discussed in discussion session) :

```
pipeline1 = Pipeline([
    ('vect', CountVectorizer(min_df=3,analyzer=stem_rmv_punc, stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('reduce_dim', NMF(n_components=50, init='random', random_state=None)),
    ('clf', MultinomialNB()),
])
pipeline2 = Pipeline([
    ('vect', CountVectorizer(min_df=3, analyzer=stem_rmv_punc, stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('reduce_dim', NMF(n_components=50, init='random', random_state=0)),
    ('toarr', SparseToDenseArray()),
    ('clf', GaussianNB()),
])
pipeline3 = Pipeline([
    ('vect', CountVectorizer(min_df=3, analyzer=stem_rmv_punc, stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('reduce_dim', NMF(n_components=50, init='random', random_state=0)),
    ('toarr', SparseToDenseArray()),
    ('clf', SVC(kernel='linear', C=10)),
])
```

Figure 7.1: Pipelines for the models

After using the pipelines, the next task was to perform 5-fold cross validation to compare the scores and accuracy.

Two types of data were extracted for this :

1. Without removing headers and footers
2. With headers and footers removed - removal of sender and last salutations

Each of these was used for grid search with 5 fold cross validation with these classifiers

1. SVM with the best gamma found (C = 10)
2. Logistic Regression with L1 and L2 regularization (C = 100)
3. GaussianNB

We also used these combinations with min_df as 3 and 5 ; with and without lemmatization

These are all the parameters from the code.

```
param_grid = [
    {
        'vect__min_df':[3,5], #min_df as 3 and 5
        'vect__analyzer': [stemmed_words,stem_rmv_punc],    #with and without lemmetization
        'reduce_dim': [TruncatedSVD(), NMF()], #two methods for dimension reduction
        'reduce_dim__n_components': N_FEATURES_OPTIONS, # reduction = 50
        'clf': [LinearSVC(C=10), #linearSVC with best gamma = 10
                LogisticRegression(penalty = 'l1', solver = 'liblinear', C=10), #L1 Logistic regression with C = 10
                LogisticRegression(penalty = 'l2', solver = 'liblinear', C=100) #L2 Logistic regression with C = 100
                ],
    },

    #Same parameters with GaussianNB and all permutations
    {
        'vect__min_df':[3,5],
        'vect__analyzer': [stemmed_words,stem_rmv_punc],    #with and without lemmetization
        'reduce_dim': [TruncatedSVD(), NMF()],
        'reduce_dim__n_components': N_FEATURES_OPTIONS,
        'clf': [GaussianNB()],
    }
]

grid = GridSearchCV(pipeline, cv=5, n_jobs=1, param_grid=param_grid, scoring='accuracy')
grid.fit(twenty_train_withoutHeaders.data, twenty_train_withoutHeaders.target)
```

Figure 7.2: Code for grid search with all combinations

The best results we got for the grid search on the two data sets is as follows :

Best Result	Model Params	Mean Test Score
With Headers and footers	Linear SVC C=10 Min_df = 5 With Lemmatization LSI	0.976

Without Headers and footers	Logistic Regression C = 10 Min_df = 5 With Lemmatization Penalty = L1 LSI	0.969

The best model comparing both the above results is - Linear SVC with headers/footers.

Test Accuracy for the models is as follows :

Test Accuracy	Model Params	Accuracy
With Headers and footers	Linear SVC C=10 Min_df = 5 With Lemmatization LSI	0.972
Without Headers and footers	Logistic Regression C = 10 Min_df = 5 With Lemmatization Penalty = L1 LSI Mind_df = 5 Penalty = L1 LSI	0.97

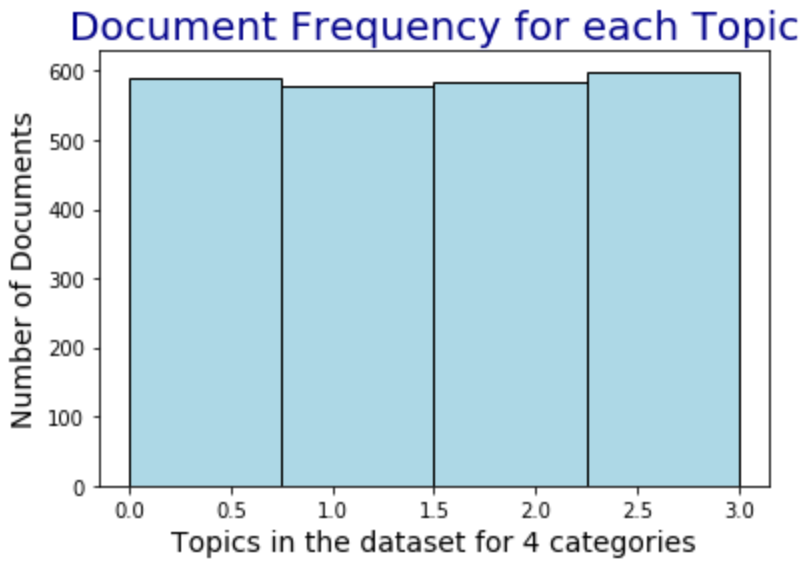
Question 8: Multiclass Classification

For this part, we first divided the training data and testing data into the four categories provided using the following code -

```
comp_categories = [ 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware']
misc_categories = [ 'misc.forsale', 'soc.religion.christian']

twenty_train = fetch_20newsgroups(subset='train', categories=comp_categories+misc_categories, shuffle=True, random_state=None,)
twenty_test = fetch_20newsgroups(subset='test', categories=comp_categories+misc_categories, shuffle=True, random_state=None,)
```

Our data when divided based on these four categories is uniformly distributed -



We then performed the count vectorization and obtained the tf-idf matrix. We again reduced the dimensions (terms) to 50. After experimenting with a variety of variables which include the type of reduction used, the gamma value for SVM, we obtained the following observations on performing the multiclass classification using naive bayes (using GaussianNB), one-vs-one SVM (using SVC with linear kernel) and one-vs-rest SVM (using LinearSVC) -

1)NAIVE BAYES CLASSIFIER -

Confusion Matrix		Predicted Labels			
		0	1	2	3
True Labels	0	272	71	37	12
	1	85	232	62	6
	2	49	37	294	10
	3	0	4	6	388

Classification Report	Precision	Recall	F1-Score	Support
0	0.67	0.69	0.68	392
1	0.67	0.60	0.64	385

2	0.74	0.75	0.75	390
3	0.93	0.97	0.95	398
Weighted avg	0.75	0.76	0.76	1565

Accuracy: 0.7578274760383387

2)ONE-VS-ONE SVM CLASSIFIER -

Confusion Matrix		Predicted Labels			
		0	**1**	**2**	**3**
True Labels	**0**	319	53	19	1
	1	47	312	25	1
	2	21	10	358	1
	3	7	1	3	387

Classification Report	Precision	Recall	F1-Score	Support
0	0.81	0.81	0.81	392
1	0.83	0.81	0.82	385
2	0.88	0.92	0.90	390
3	0.99	0.97	0.98	398
Weighted avg	0.88	0.88	0.88	1565

Accuracy: 0.8792332268370607

3) **ONE-VS-REST SVM CLASSIFIER -**

Confusion Matrix	Predicted Labels

		0	1	2	3
True Labels	0	315	52	21	4
	1	40	324	19	2
	2	19	14	356	1
	3	3	2	2	391

Classification Report	Precision	Recall	F1-Score	Support
0	0.84	0.80	0.82	392
1	0.83	0.84	0.83	385
2	0.89	0.91	0.90	390
3	0.98	0.98	0.98	398
Weighted avg	0.89	0.89	0.89	1565

Accuracy: 0.8856230031948882

We observe that the one-vs-rest SVM classifier has the best accuracy on the testing data.