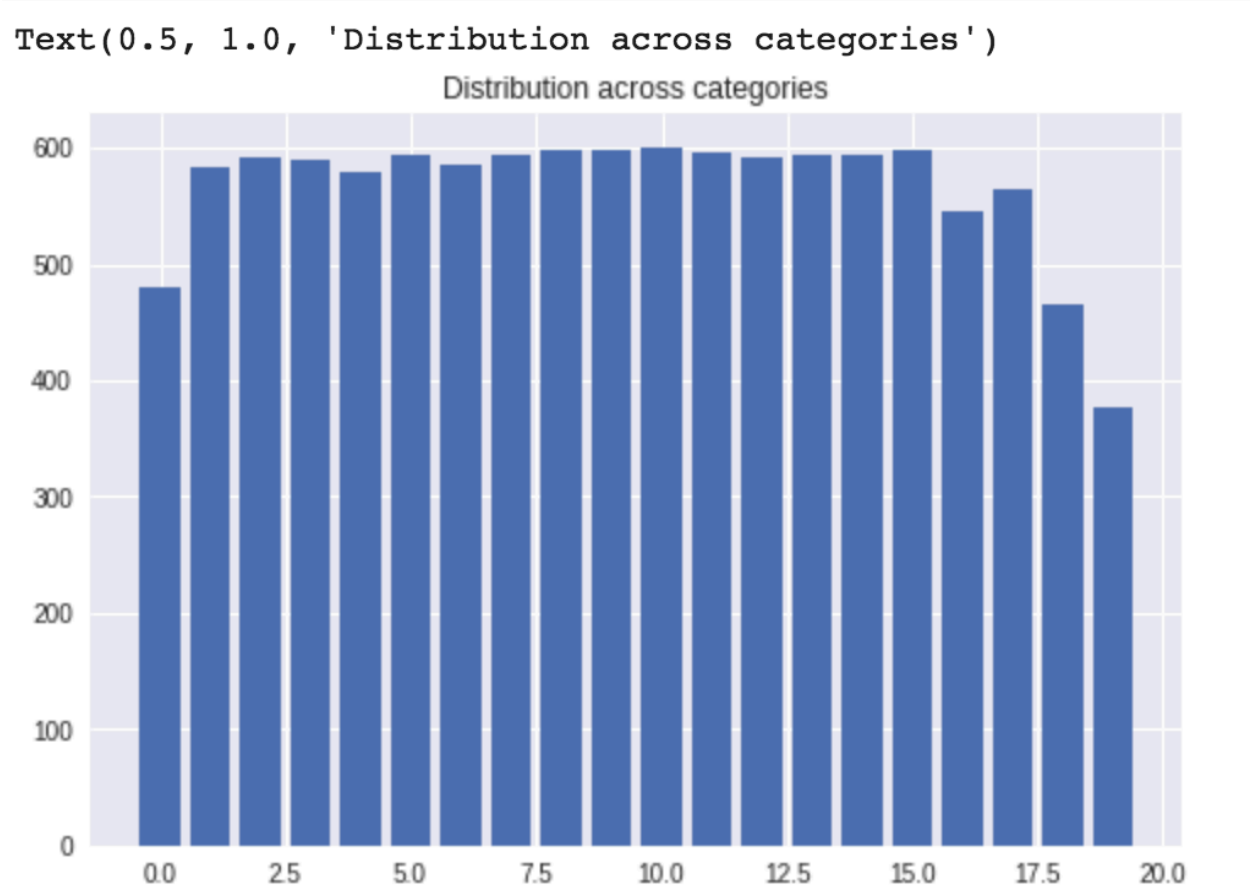
Introduction

In this project we have classified documents from the "20 Newsgroup Dataset" based on textual data. We compare different classification techniques in conjunction with various data cleaning and normalization methods and compare their results. We have used confusion matrix and scoring parameters like accuracy, F- 1, recall and precision to draw a comparison between the various methods used.

Question 1

Before we start with the classification we need to check if the data is evenly distributed so as to avoid skewed learning. To validate the distribution, we create a histogram of the class label from the dataset against number of documents. If the distribution is skewed, we can penalize the over-represented classes, or modify the weights to create an evenly distributed dataset.

We see that the distribution is evenly distributed enough to continue with our classification without any penalization or modification. Also, it was explicitly stated that the class labels we would be working with in this classification experiment was uniformly distributed.



Question 2

Once we have completed analyzing the distribution, we move on to cleaning and extracting features from the textual data. One of the basic steps is to drop the "English" stop words and punctuation from the documents, as they add no value to the classification and can have high frequency thus unnecessarily increasing the complexity.
To standardize the words across the documents we convert every word to its lemma (canonical form), for eg. run, runs, ran and running has run as the lemma, thus they will be converted to that. Since the meaning of the word is conveyed using the lemma and also allowing us to check for its frequency across all documents this is a reasonable way to reduce the dimensionality of the data.

```
lemmatize_sent_demo('He is walking to school')
```

```
['he', 'be', 'walk', 'to', 'school']
```

```
print(X_train_counts.shape)    (4732, 16600)
print(X_test_counts.shape)     (3150, 16600)
print(X_train_tfidf.shape)     (4732, 16600)
print(X_test_tfidf.shape)      (3150, 16600)
```

Stemming and lemmatization

For grammatical reasons, documents are going to use different forms of a word, such as organize, organizes, and organizing. Additionally, there are families of derivationally related words with similar meanings, such as democracy, democratic, and democratization. In many situations, it seems as if it would be useful for a search for one of these words to return documents that contain another word in the set.
The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. For instance:

am, are, is $\Rightarrow$ be

car, cars, car's, cars' $\Rightarrow$ car
The result of this mapping of text will be something like:

the boy's cars are different colors $\Rightarrow$ the boy car be differ color

However, the two words differ in their flavor. Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma.

Reference : https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

We also drop all the numerical values in the documents as the value by itself does not help us classify in our case. The topic what the value refers is to of more importance to us.

After cleaning the data and standardizing it across all documents we calculate the "Term Frequency-Inverse Document Frequency (TF-IDF)" metric of the terms present in every document. The tf-idf value quantifies if the term is a discriminating term for the document, words which are present across multiple documents are not distinguishing feature and thus are given a smaller value compared to those words which might help in classifying the document.

The tf-idf(d,t) is given by

$$tf - idf(d,t) = tf(t,d) * idf(t)$$

where

$$idf(t) = log(\frac{n}{df(t)}) + 1$$

After calculating the TF-IDF matrix using scikit.CountVectorizer() we get:

Training Data of Shape: (4732,16600)
Testing Data of Shape: (3150, 16600)

Question 3

Once the features have been calculated using CountVectorizer, we perform dimensionality reduction using feature projection.
When we keep adding features without increasing the number of training samples as well, the dimensionality of the feature space grows and becomes sparser and sparser. Due to this sparsity, it becomes much easier to find a "perfect" solution for the machine learning model which highly likely leads to overfitting. Feature projection tries to select a subset of the original features for use in the machine learning model. In this way, we could remove redundant and irrelevant features without incurring much loss of information. Feature projection transforms the data in the high-dimensional space to a space of fewer dimensions using linear or nonlinear techniques.
In our project we compare two feature reduction techniques: Latent Semantic Indexing (LSI)
and Non-negative Matrix Factorization (NMF).

Latent Semantic Indexing casts queries into this low-rank representation, enabling us to compute query-document similarity scores in the low-rank representation. This vector space representation enjoys a number of advantages including the uniform treatment of queries and documents as vectors, the induced score computation based on cosine similarity, the ability to weight different terms differently, and its extension beyond document retrieval to such applications as clustering and classification.

NMF, in its most general form, can be described by the following factorization

$$Xd \times N = Wd \times r\ Hr \times N + E$$

where d is the dimension of the data, N is the number of data points (usually more than d) and r<d, E is the error and X, W, H are non-negative.

Upon performing feature reduction to 50, we calculate the error using mean-squared error and compare the performance.

NMF mean squared residual : 3937.744
LSI mean squared residual : 3895.33

The reason NMF performs worse than LSI is due the extra constraint of keeping the matrix positive which leads to loss of data.

Reference :
https://nlp.stanford.edu/IR-book/html/htmledition/latent-semantic-indexing-1.html

Some definitions and formulae for precision, recall, and f-measure used for computing the correctness of the models.

Precision: exactness- what percentage of tuples that the classifier labeled as positive are actually positive.

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$= \frac{True\ Positive}{Total\ Predicted\ Positive}$$

Recall: Completeness- what percentage of positive tuples did the classifier label as positive.

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

$$= \frac{True\ Positive}{Total\ Actual\ Positive}$$

F1 Score is a function of Precision and Recall. A measure that combines precision and recall is the harmonic mean of precision and recall

$$F1 = 2 \times \frac{Precision * Recall}{Precision + Recall}$$

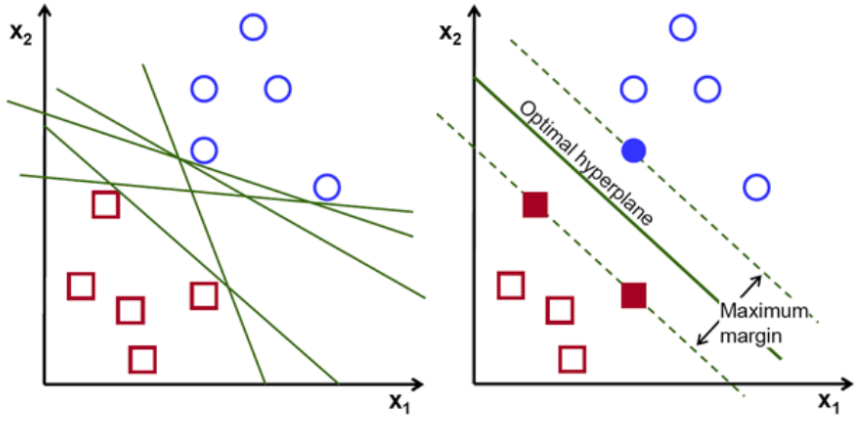
ROC Curve: The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The greater the area under the curve, the better performance the classifier has.

Confusion matrix: Each **row** of the matrix represents the instances in a **predicted class** while each **column** represents the instances in an **actual class** (or vice versa).

Question 4

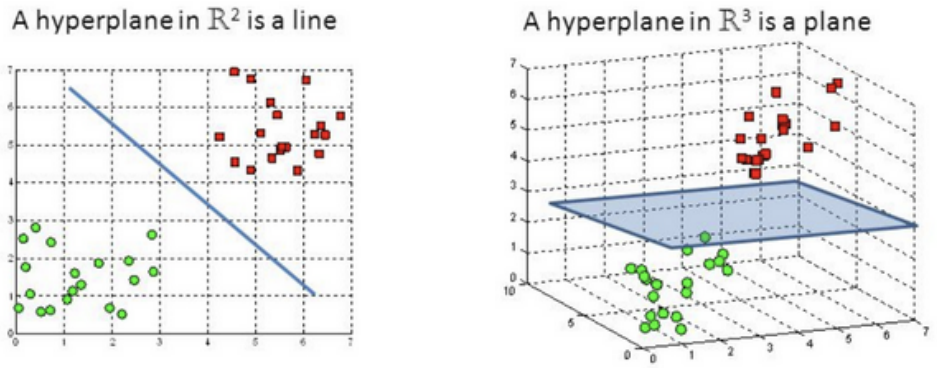
SVM

In SVM, the objective is to find a hyperplane in N dimensional space that would correctly classify the point. However, the problem is there can be many hyperplanes that classifies the data-points, but SVM finds the best hyperplane. Points that lie on one side of the hyperplane will be classified to one class while points that lie on the other side will be classified to a different class.



Possible hyperplanes

The optimal hyperplane is obtained by the maximum margin which maximizes the distance between the data-points of both classes. The reason we do this is so that we can have more confidence in classifying. The dimensions of the hyperplane depend on the number of features of our input. If our input has two features, the hyperplane is just a line, while if it has three features, it becomes a two-dimensional plane.



Hyperplanes in 2D and 3D feature space

The data-points that define a hyperplane is called support vectors. These points are those that are closer to the hyperplane and will actually be the only points that will determine the hyperplane. This means that if we remove or change these points, our hyperplane will shift. Conversely, removing other points that are not support vectors will not change the hyperplane at all.

The decision boundary of SVM is decided by thresholding a linear equation.

Example:

$$sign(\mathbf{w}^{\mathsf{T}}\mathbf{x} + b)$$

If the value is negative, we assign data point to one label, while if the value is positive, we assign the data point to another label. In order to find the optimal hyperplane (w*, b*), it is just solving an optimization problem. Let's first assume that all our data is linearly separable. We'll be solving the problem below:

Primal Form

Objective: Find $\mathbf{w}$ and b such that $\Phi(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w}$ is minimized;

Constraints: for all $\{(\mathbf{x_i}, y_i)\}$: $y_i(\mathbf{w}^T\mathbf{x_i} + b) \geq 1$

Solving the dual form equals solving the primal form in this case, and it is actually easier.

Dual Form

Objective: Find $\alpha_1 \ldots \alpha_n$ such that

$\mathbf{Q}(\alpha) = \Sigma\alpha_i - \frac{1}{2}\Sigma\Sigma\alpha_i\alpha_j y_i y_j \mathbf{x}_i^T\mathbf{x}_j$ is maximized and

Constraints

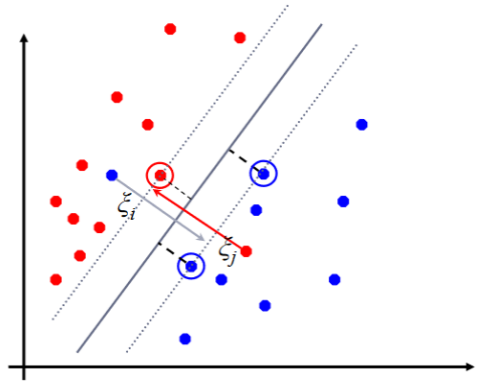
(1) $\Sigma\alpha_i y_i = 0$

(2) $\alpha_i \geq 0$ for all α_i

After differentiating the equation with respect to w and b, we get the solution as below:

$\mathbf{w} = \sum\alpha_i y_i \mathbf{x_i}$ $\quad b = y_k - \mathbf{w}^T\mathbf{x_k}$ for any $\mathbf{x_k}$ such that $\alpha_k \neq 0$

Each non-zero alpha corresponds to support vectors. From the equation we can see that only the points that are support vectors have effect on the hyperplanes.



In this case we have assumed that the data points are linearly separable which is usually not the case. In our homework, the data points are also not linearly separable. This is why we introduce another kind of SVM called soft margin SVM. In soft margin SVM, we allow some misclassification with a cost assign to those misclassified points. Intuitively, it is like moving the points so that they are correctly classified.

By introducing gamma (also known as slack variable), the dual problem becomes:

Find $\mathbf{w}$ and b such that

$\Phi(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum\xi_i$ is minimized and for all $\{(\mathbf{x_i}, y_i)\}$

$y_i(\mathbf{w}^T\mathbf{x_i} + b) \geq 1 - \xi_i$ and $\xi_i \geq 0$ for all i

After solving the problem, we got the values of w* and b* as below:

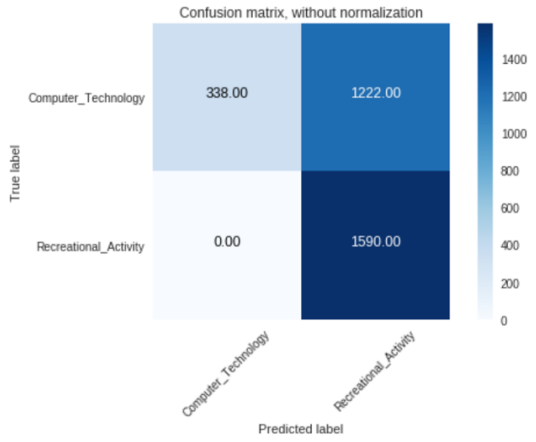
$\mathbf{w} = \sum\alpha_i y_i \mathbf{x_i}$

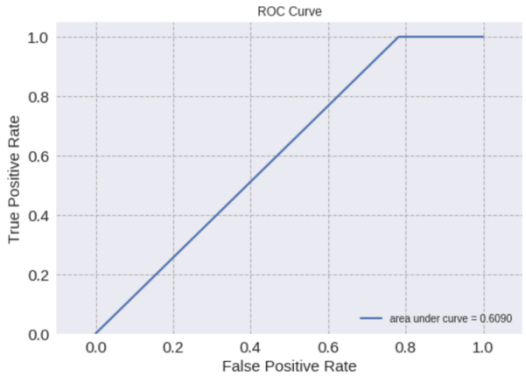
$b = y_k - \mathbf{w}^T\mathbf{x_k}$ for any $\mathbf{x_k}$ such that $0 < \alpha_k < C$

Again, all the alphas that are non-zero are those that are support vectors. We can also view this optimization problem as a loss function where we optimize our parameters over a loss. The loss is known as the hinge loss equation which is given below:

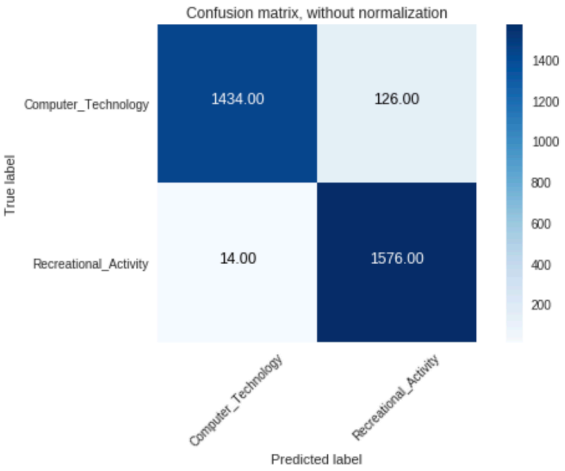
$$\min_{\mathbf{w}\in\mathbb{R}^d} \underbrace{||\mathbf{w}||^2}_{\text{regularization}} + C\sum_i^N \underbrace{\max(0, 1 - y_i f(\mathbf{x}_i))}_{\text{loss function}}$$

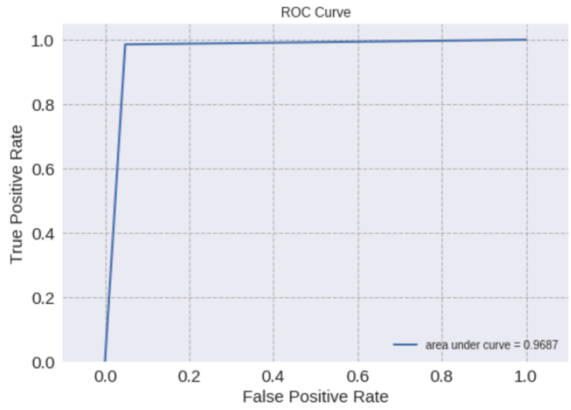
Results from soft margin SVM:
Accurcy = 0.6120634920634921
Precision = 0.5654338549075392
Recall = 1.0
F1 Score = 0.7223989095865516





Results from hard margin SVM:
Accuracy = 0.9555555555555556
Precision = 0.9259694477085781
Recall = 0.9911949685534591
F1 Score = 0.9574726609963548



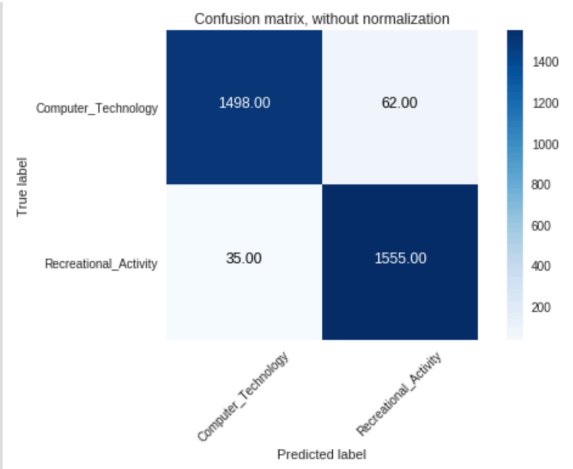


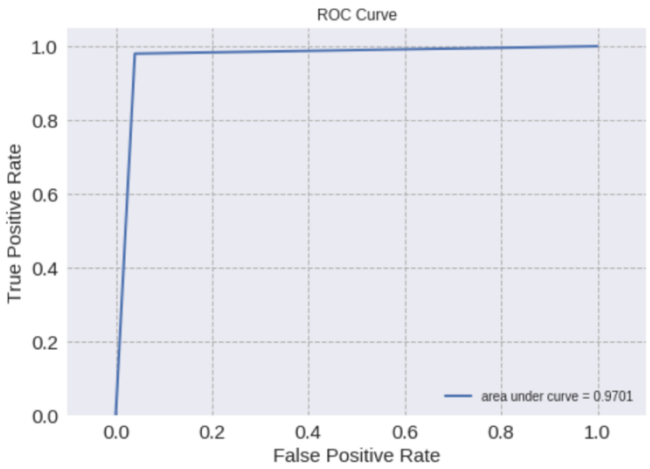
What happens for the soft margin SVM? Why is the case?
* Does the ROC curve of the soft margin SVM look good? Does this conflict with other metrics?

In this part, we need to compare the results from hard margin SVM (C=1000) and soft margin SVM (C=0.0001). We can see that in almost all of the categories, hard margin SVM performed better than soft margin SVM. The reason that hard margin SVM performed well is because the data is probably linearly separable. If you look at the categories we're trying to classify, it's dividing the data into recreational activities and computers. The text probably contains very different words that is why hard margin SVM performed really well. Now, intuitively, soft margin SVM should be able to outperform hard margin SVM. However, since the data is almost linearly separable, by setting the wrong C values, we might actually cause more problem because we're allowing room for more errors. We also tried playing around with other C values. If we actually tune the C up to 0.01, it actually did a lot better than 0.0001. We can also see that the ROC curve for soft margin SVM is very bad which is also shown in a low accuracy, precision, and F1 score. The reason that recall is 1.0 in soft margin SVM is because we do not have any false negative. In this case, most of the time we predict the labels to be Recreational Activity making the recall to become 1.

Now, instead of fixing a C value, we will do a grid search over few values of C. We found that the best value of C is equal to 10. The evaluation metric we used to find C is based on the best accuracy on the training dataset. The results are as below.

Accuracy = 0.9692063492063492
Precision = 0.9616573902288188
Recall = 0.9779874213836478
F1 Score = 0.9697536638603057





We can actually see that soft margin SVM did perform better in the accuracy metrics than the hard margin SVM. However, the difference is not much since the data is already almost linearly separable. We learn that if the data is linearly separable, by trying to allow more room for errors (setting a low value for C), it might actually result in a worse performance than just using a hard margin SVM.

Question 5

Logistic Regression

Logistic regression is a type of regression that returns a probability instead of a continuous value. In case of binary classification, we predict the class label as the positive class if the probability is greater than 0.5 and to the negative class if it is lesser than 0.5. Logistic regression is based on the sigmoid function acting on a linear function as written in the formula below:

where

$$\sigma(\phi) = 1/(1+\exp(-\phi))$$

$$\phi(\mathbf{x}) = \mathbf{w}^{\mathsf{T}}\mathbf{x}+b$$

In order to find the parameters of the equation, we decided to use the built-in solver from Scikitlearn which is liblinear. What liblinear does is it does a coordinate descent in order to find the best parameters. We decided to use it because It is said to work pretty well on binary classification of data with high dimensions. From the SVM part, we can see that the data is very close to being linearly separable. This is why we expect that logistic regression will perform really well because in logistic regression, we are also using a linear boundary.

First, we have to implement logistic regression without any penalty term. However, in Scikitlearn, there is no option for setting no penalty. This is why we choose to use L2 penalty and set C to a very large number. According to the documentation, C is inversely proportional to lambda, so if we set it to a large value, we can think of it as setting the regularization term to 0 which reverts back to a normal logistic regression. In this question, we decided to use C = 1e60 and L2 Penalty Cost function to find w of logistic regression. If we set lambda to 0, we return to the normal cost function without any penalty.

$$J(\mathbf{w}) = \sum_{i=1}^{n}\left[-y^{(i)}\log\left(\phi(z^{(i)})\right) - \left(1-y^{(i)}\right)\log\left(1-\phi(z^{(i)})\right)\right] + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

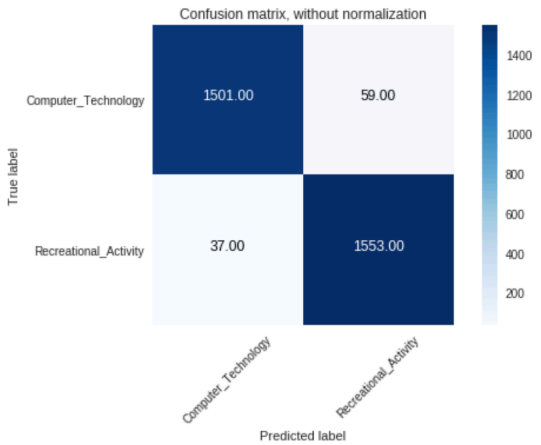
The results of L2 without regularization are as below:

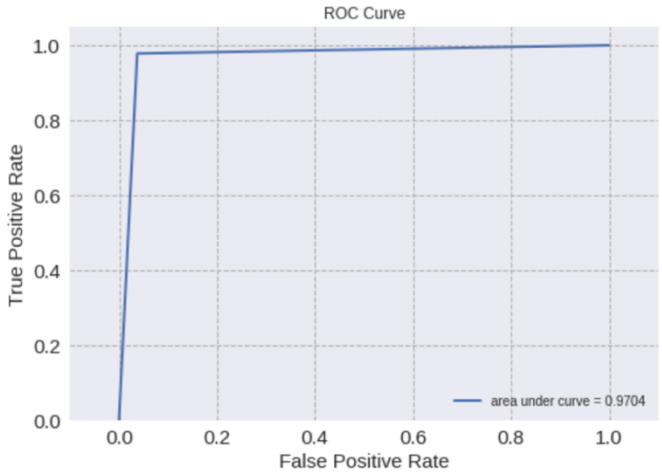
Accuracy = 0.9695238095238096

Precision = 0.9633995037220844

Recall = 0.9767295597484277

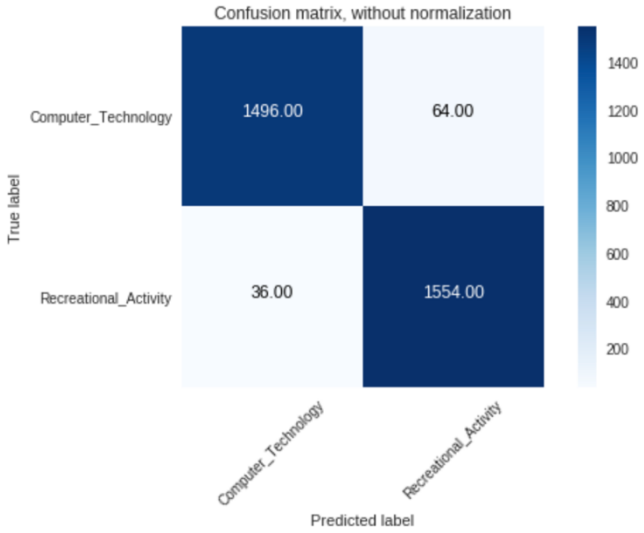
F1 Score = 0.9700187382885697

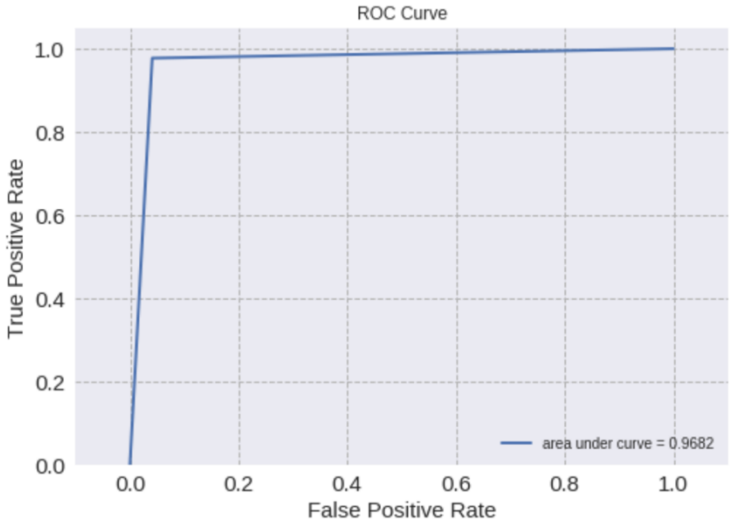




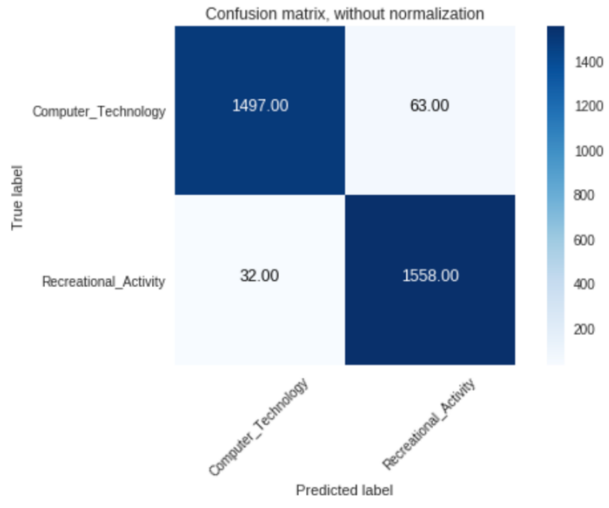
In L1 Penalty, instead of taking the squares of each element and summing them, we take the absolute value and sum them up instead. What these penalty term does is it acts as a regularization to help solve the problem of overfitting to the training data. We're trying to get our weights to become small since if the weight is large, we might be overfitting and we do this by adding in the penalty or regularization term. It is noticeable from the equation that if our weights our large, the cost will increase.

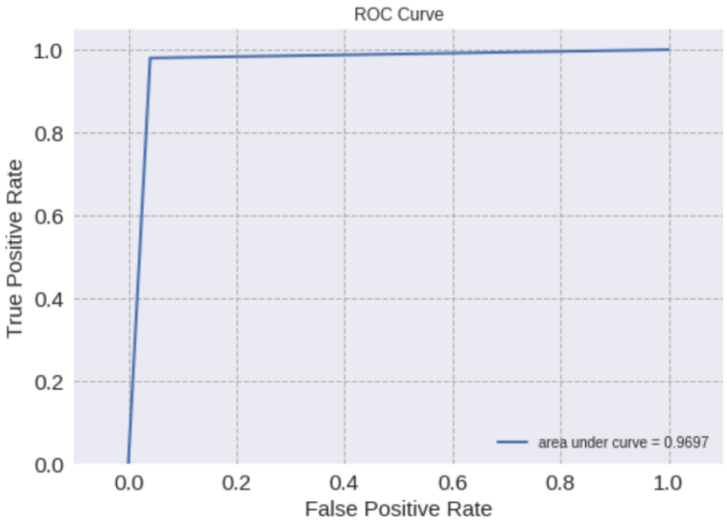
The results of L1 Penalty with regularization are as below:
Accuracy = 0.9682539682539683
Precision = 0.9604449938195303
Recall = 0.9773584905660377
F1 Score = 0.9688279301745636





The results of L2 Penalty with regularization are as below:
Accuracy = 0.9698412698412698
Precision = 0.9611351017890192
Recall = 0.979874213836478
F1 Score = 0.970414201183432





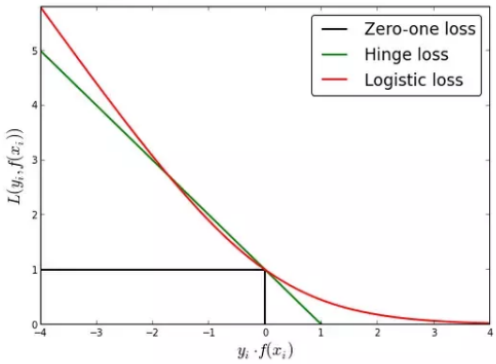
In order to get the best C, we do a grid search over the parameters with the evaluation metrics as accuracy.

How does the regularization parameter affect the test error? How are the learnt coefficients affected? Why might one be interested in each type of regularization?

We can see that the L1 and L2 Penalty does not have much effect in this case. This is probably because the data is already well-defined meaning that the training resembles the testing pretty well. Normally, the L1 and L2 penalty increase the bias but reduce the variance of the data since without these parameters we might be overfitting to the training data. If you actually look at the coefficients, it is actually smaller when you use L1 and L2 penalty which is as expected. Now when do we want to use L1 or L2 penalty? If you think about it in terms of probability, we can think of these regularization terms as our prior knowledge. If we believe that our data is from a Gaussian distribution, we can use L2 penalty because that is modeled from that kind of distribution. However, if we believe that our data is from a Laplace distribution, we will use L1 penalty. However, overall, we would use L2 penalty over L1 penalty almost all the times.

Both logistic regression and linear SVM are trying to classify data points using a linear decision boundary, then what's the difference between their ways to find this boundary? Why their performance differ?

First of all the loss function for SVM and logistic regression is different. In SVM, we're using a hinge loss while in logistic regression we're using a logistic loss. If we actually plot these out, we will see that logistic loss actually diverges much faster than hinge loss making it more prone to outliers.



Loss functions

Another intuition from this is that logistic loss provides a probabilistic point of view. Even though it is confident, logistic loss does not go to zero making accuracy lower sometimes. With respect to the decision boundary, SVM tries to find the maximum margin for the points. However, logistic regression tries to find the maximum posterior probability. Thus, SVM will find a solution that divides the two classes as far as possible while logistic regression does not have this property making SVM able to outperform logistic regression in most cases. In this case since the data is already almost linearly separable, the two classifiers perform similarly.

Question 6

In this part we try another classifier called Naive Bayes to separate the data into two classes "Computer Technology" vs "Recreational Activity". Naive Bayes Classifiers are a family of probabilistic classifiers based on applying Bayes' Theorem with strong (naive) assumptions that each of the features used for classification are independent of each other.

Naive Bayes is a conditional probability model: given a problem instance to be classified, represented by a vector representing some n features (independent variables), it assigns to this instance probabilities for each of K possible outcomes or classes

Using Bayes' theorem, the conditional probability can be decomposed as

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k)\, p(\mathbf{x} \mid C_k)}{p(\mathbf{x})}$$

Using Bayesian probability terminology, the above equation can be written as

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{evidence}}$$

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of P(xi|y).

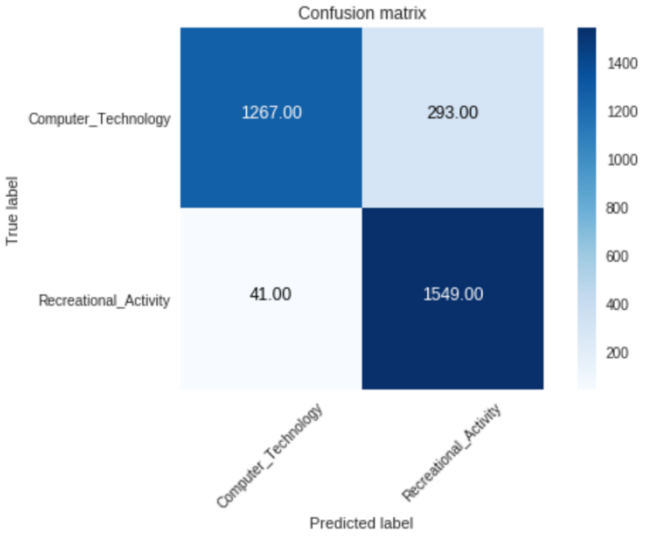
When dealing with continuous data, a typical assumption is that the continuous values associated with each class are distributed according to a Gaussian distribution. Since the likelihood of the features is assumed to be Gaussian, therefore we implement the Gaussian Naive Bayes Algorithm for the following problem.

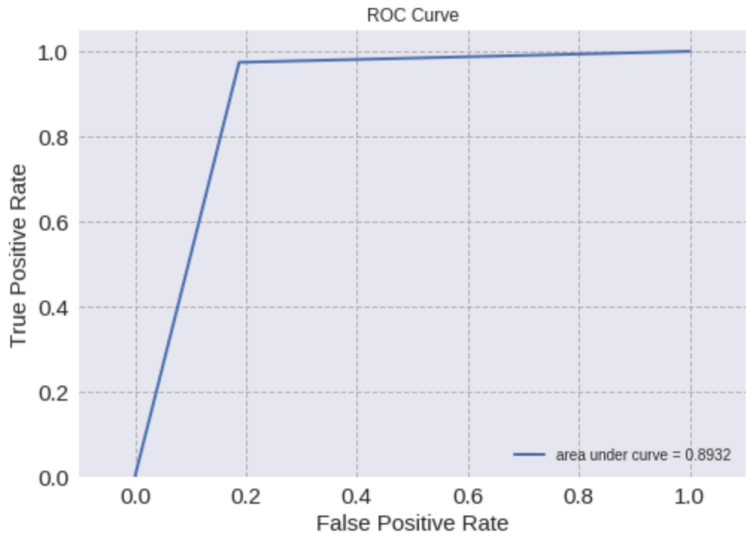
$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

Scikit-learn provides a type of classifiers called “naïve Bayes classifiers”. They include MultinomialNB, BernoulliNB, and GaussianNB. We use the GaussianNB function to perform the classification on our data.

We fit the model to the Trained dataset that has been reduced by performing Latent Semantic Analysis (LSI) comparing it with the labels from the training dataset that we got after combining the documents into the two categories. Using the classifier object created, we call the predict function on the test dataset (which has also been reduced by LSI). The predicted result is then compared with the actual labels from the test dataset.

The results obtained are as follows:
Accuracy = 0.893968253968254
Precision = 0.8409337676438654
Recall = 0.9742138364779874
F1 Score = 0.9026806526806527





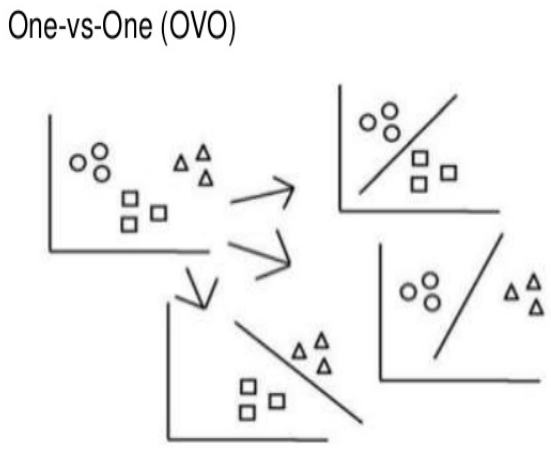
This shows that our classifier was able to classify our data into the two respective categories with good accuracy and precision. Since the area under the curve also gives us a high performance rate, we can conclude that our Gaussian Naive Bayes classifies our data correctly.

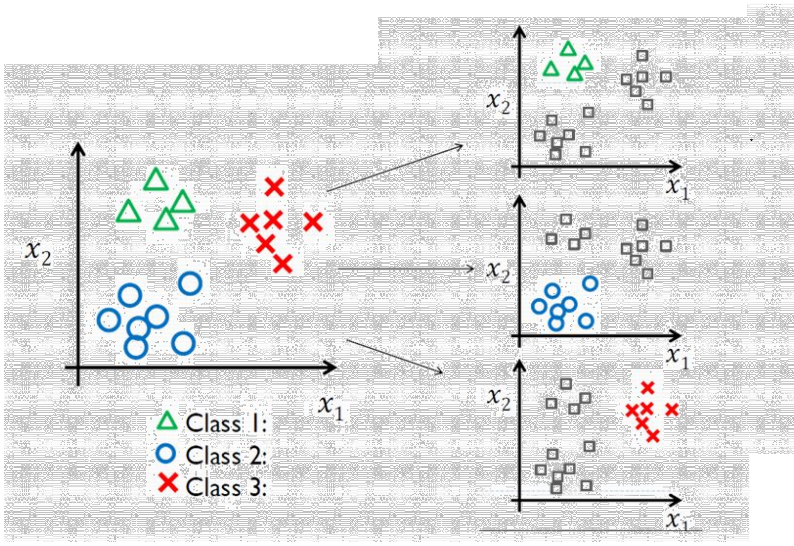
Question 7

After performing classification over our dataset we see that a number of parameters can be tuned to get better results for the classification problem. We then create a pipeline to perform all the tasks in order. For this problem we use the Pipeline function from the Scikit Learn library. We perform vectorisation, tf-idf transformation, dimensionality reduction and classification inside the pipeline. We create a temporary memory cache to store all our data. To get the best parameters we perform a grid search with 5-fold cross-validation to compare the different parameters using the test accuracy as the score to compare. We define the various permutations and pass it into the grid search.
The min_df argument of the CountVectorization function is passed with the values 3,5.
The analyzer argument is passed with lemmatization and without.
The dimensionality reduction is tried with both methods LSI as well as NMF.
And all these parameters are tuned across each of the three classifiers SVC, Logistic Regression (with L1 and L2 normalization) as well as GaussianNB. The SVC and Logistic regression classification is performed with the cost function C=10 (as found earlier with the grid search) and the Naive Bayes classifier doesn't have a cost function explicitly defined.
We load the data separately, one with headers and footers and one without. The headers and footers are removed from the docs while loading the data by using the remove option. Grid search is performed separately on both the datasets and the results are compared as follows.
After performing grid search, the best combination of all the parameters were found in the Logistic Regression classifier with L1 normalization having cost function C=10. The classifier had higher accuracy with headers and footers included, with lemmatization. The dimension reduction technique used was LSI with n_components being 50 i.e the size of the vector in reduced space. The document frequency is set to threshold min_df = 3 for the best accuracy.
If you want the explicit data results for each of the possible combinations of parameters with header/footer and without header/footer then please check the python notebook for grid1.cv_results_ and grid2.cv_results_.

Question 8

The Naïve Bayes algorithm generally supports multiclass classification directly by choosing the class with the highest probability. However, in SVM, there is no support for multiclassification. This is why we introduced two techniques known as one vs one and one vs rest. In one vs one, we train a $\frac{N(N-1)}{2}$ classifiers where N is the number of classes we have. Another technique is one vs rest where we would have N classifiers. We choose one class as our classifier and assume all the other classes are classes that is not the class we choose. These are illustrated in the picture below:





We trained a linear SVM with these two techniques and also performed a Gridsearch over the C values. We have also decided to use macro average to find the recall, precision, and F1 score of the multiclass classifications. Macro calculates metrics for each label, and find their unweighted mean. This does not take label imbalance into account. This is because most of our classes our already balance which is why we think this method would work well.

We can see from our result that one vs one performs better than one vs all. There is no specific reason why one would be better than another. Next, we also do a Gaussian Naïve Bayes classifier to do multiclass classification. The reason we use Gaussian is the because our X_train values are continuous, and Gaussian is one that supports continuous values. This gives the result below:

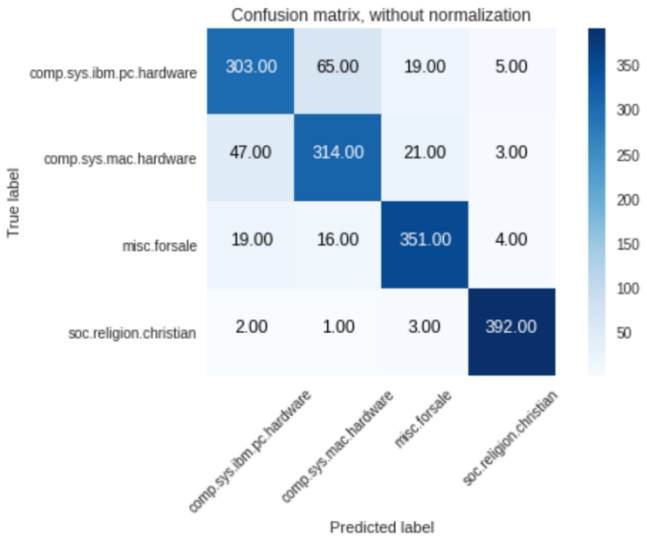
We can see that SVM performs better than Gaussian Naïve Bayes, and this might be because Gaussian Naïve Bayes assumes that the data are independent. However, SVM sorts of still maintain those dependencies. This is why sometimes, SVM will perform better than Gaussian Naïve Bayes in this case.

print(X_train_counts.shape). (2352, 8699)
print(X_test_counts.shape) (1565, 8699)
print(X_train_tfidf.shape) (2352, 8699)
print(X_test_tfidf.shape) (1565, 8699)

LSI mean squared residual : 1800.7489051066314

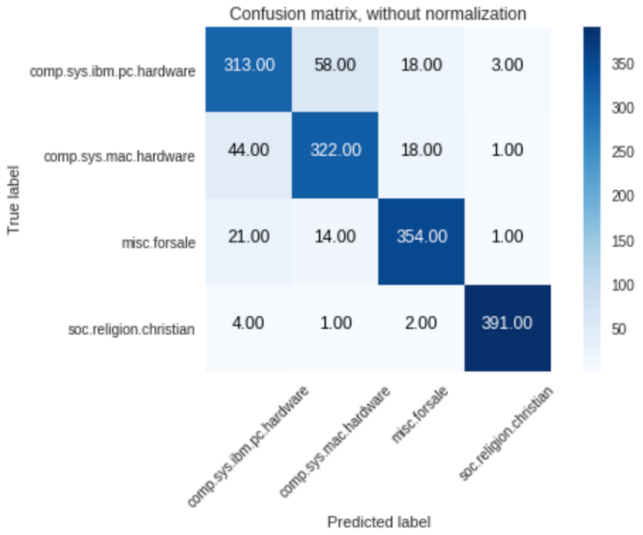
Grid search for best C in multiclass linear SVM using one vs rest technique with precision and recall as macro

Accuracy = 0.8690095846645367
Precision = 0.867700214272799
Recall = 0.8683670555933658
F1 Score = 0.8678237184441725



Grid search for best C in multiclass linear SVM using one vs rest technique with precision and recall as macro

Accuracy = 0.8817891373801917
Precision = 0.8812491407574865
Recall = 0.8812343480281384
F1 Score = 0.8811710123051533



Multiclass Gaussian Naive Bayes with precision and recall as macro

Accuracy = 0.7361022364217252
Precision = 0.7385407062019868
Recall = 0.7337629721917232
F1 Score = 0.7197683150959924