Assignment - 2

Samyak Jain(2019098) and Sarthak Johari(2019099)

Question - 1

- Jaccard Coefficient

Preprocessing :

We have used the ‘os’ library of python to traverse and pick files from the data directory. The text of each file is read and is preprocessed and finally, we have tokens corresponding to each file.

Preprocessing steps: (Same as mentioned in A1)

1. The text is converted to lowercase.
2. Word tokenization is performed on the text.
3. Stopwords are removed from the tokens.
4. Punctuation marks have been removed from the tokens.
5. Blank space has been removed from the tokens.

These operations are performed in order for all the text of all the files to get a list of tokens for each file.

Methodology

In this part, we calculate the Jaccard coefficient between a given query and the document. The higher the value of the Jaccard coefficient the more the

document is relevant for the query. The Jaccard coefficient is calculated using the formula :

Jaccard Coefficient = Intersection of (doc,query) / Union of (doc,query)

This is basically the ratio of the number of terms in the intersection of the token sets of the document & query and the number of terms in the union of the token sets of the document & query. The value of the Jaccard coefficient is between 0 and 1.

To compute the Jaccard coefficient between a query and a document, we firstly make the set of document tokens and query tokens for that document and the query. Then We calculate the intersection of these sets as well as the union of these sets. The value of the Jaccard coefficient is the ratio of the cardinality of the intersection set to the cardinality of the union set.

For a given query, the value of the Jaccard coefficient is calculated for each query, document pair and finally, we sort them in descending order to get the top 5 documents with the highest Jaccard coefficient values.

- TF-IDF Matrix

We use the same preprocessed data we have in the first part (Jaccard coefficient one).

Methodology

We form the total vocabulary set using the individual document-wise tokens we have after preprocessing. Also for matrix creation purposes, we map all the terms to a term id because in matrices we refer to individual cells by a pair of indices which would indicate (document id, term id) pairs.

For calculating the TF-IDF Matrix, we have to calculate two things, term frequency(TF) and inverse document frequency (IDF). Firstly for calculating

the term frequency, we find out the raw count of terms in each document. This is stored as a nested dictionary, where for the outer dictionary, the key is the document-id and the value is another dictionary that stores the terms and their counts for the terms present in that document. Through this, we get the raw term frequency of each term present in each document. Now for calculating the IDF, we first calculate the document frequency (DF) for each term in the vocabulary. To find this, we form the posting lists for each term in the vocabulary and the length of the posting list is equal to the document frequency for the term. With the document frequency of each term calculated, we find the IDF for each term using the formula :

$$IDF_{term} = \log(N_{docs} / (DF_{term} + 1))$$

where N_{docs} is the total no. of documents in our dataset, and DF_{term} is the document frequency of the term.

Now, we form the TF-IDF matrix for each of the 5 TF weighting schemes:

Weighting Scheme	TF Weight
Binary	0,1
Raw count	f(t,d)
Term frequency	$f(t,d)/\sum f(t', d)$
Log normalization	$\log(1+f(t,d))$
Double normalization	$0.5+0.5*(f(t,d)/ \max(f(t',d))$

The TF-IDF matrix is a 2-dimensional matrix of size (No. of docs x no. of terms in vocab). The $(i, j)^{th}$ element of the matrix is equal to the TF-IDF value of the term with the term id ‘j’ in the document id ‘i’. The value in the cell is equal:

$$Weight(TF_{j,i}) \times IDF_j$$

where $TF_{j,i}$ is the value of term frequency of term with term id ‘j’ in the document with document id ‘i’.

In this way, five different TF-IDF matrices are generated one for each weighting scheme.

Processing the query

When we get the query, we first preprocess the query using the same steps as we used for document preprocessing and get the query tokens. Now we construct a query vector for the query. It is the size of the number of terms in the vocabulary set. We implement the TF-IDF variant where the query terms are also weighed in the same manner as the document. The term frequencies of the terms in the query are also calculated. No for each term in vocab, we find that word in the query and take its term frequency in the query, We multiply the weight of this term frequency(using the scheme we used for calculating the TF-IDF matrix) with the IDF of the term and fill this value in the query vector at the index corresponding to the term ID. This way we form the query vector.

We take the dot product of the (N_{docs} x |v|) TF-IDF matrix with the (|v| x 1) query vector [N_{docs} = Total no. of documents, |v| = size of vocabulary] to get a resultant (N_{docs} x 1) vector which has the TF-IDF score for the query with each document. We sort this in descending order to get the top 5 documents with the highest score and report them as the top 5 relevant documents.

This process is done for each weighting scheme and the results for each weighting scheme are reported for a given query.

Assumptions:

- If we get an empty query, we don't process anything (program terminates) and return the user a message that the query is empty.
- The query is preprocessed using the same preprocessing steps as we have to preprocess the documents to get the query tokens.

- If a term is not present in the document, i.e - the TF value of the term in the document is zero, then the value of the weight of the TF is also zero for all weighting schemes except the Double Normalization scheme where according to the formula the value of TF weight in that case is 0.5.
- If there is any tie in some documents wrt to the Jaccard Coefficient score or the TF-IDF score, any document can be returned as more relevant than the other. There is no precedence scheme followed to manage ties.

Pros and Cons of each technique:

- Jaccard Coefficient
  - + Simpler technique assigns a score between 0 to 1 always.
  - + The query representation (query tokens set) and document token set need not be of the same size.
  - - It does not consider the term frequency at all for scoring purpose. It is only affected by the mere presence/absence of the term in the query, document token set.
  - - Jaccard Coefficient doesn't consider the information regarding rare terms being more informative than frequent terms.
  - - It does not consider the ordering/position of words in the document (but it is important as changing the position of words transforms the semantic meaning of a sentence for eg. A is better than B – VS – B is better than A).

- TF-IDF Matrix based
  - + Takes into account the frequency of terms in a document (with the inclusion of the term frequency part).
  - + Consider the information regarding rare terms being more informative than frequent terms (by including the IDF value). High weights are given to rare terms.
  - - It also does not consider the ordering of words/terms in a document.
  - - Can be slower for large-sized vocabularies.
  - - Also, it does take into account any kind of semantic similarity.

Question - 2

1. The dataset had many query-URL pairs along with their relevance judgement score. We firstly extract only those queries with qid:4. We get 103 such queries. Their relevance judgement scores (which is the first integer in each query line) are taken as the relevance scores for them. Now we have the queries with qid:4 and we'll use this for further parts.

2. To make a file with query-URL pairs arranged in the order of max DCG, we know that we will have to arrange them in the decreasing order of their relevance scores. Thus we do the same to make an arrangement in which the query-URL pairs are arranged in the decreasing order of their relevance scores and this arrangement will correspond to the max DCG arrangement. We form one such file have this kind of arrangement. Now to calculate how many such files can be made, we find the counts of the query-url pairs corresponding to each relevance score. They are as follows:

Counts wrt labels: {0: 59, 1: 26, 2: 17, 3: 1, 4: 0}

Thus, as we need decreasing order of arrangement the number of possible files with such arrangement = 0! * 1! * 17! * 26! * 59! which is equal to 19893497375938370599826047614905329896936840170566570588205180312704857992695193482412686565431050240000000000000000000000000.

3. For calculating the DCG, we use the following formula :

$$DCG_p = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2 i}$$

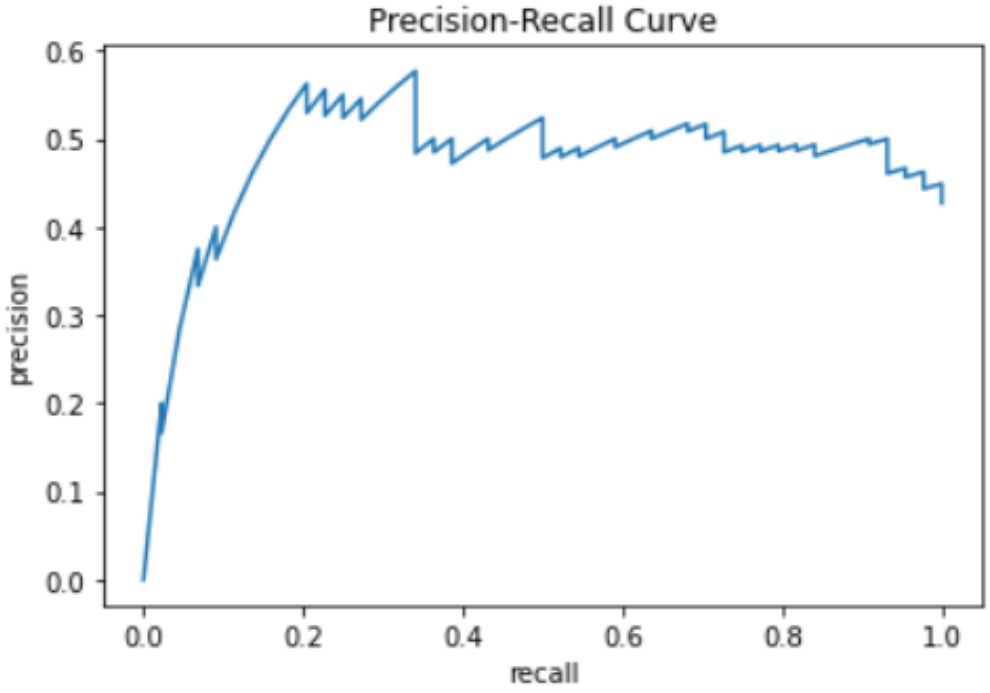
where DCG_p is the discounted cumulative gain at rank p and rel_i is the relevance score for the document (here query-URL pair) with rank 'i'. For finding nDCG at 'p' we use the formula :

$$nDCG_p = DCG_p \ / \ IDCG_p$$

where DCG_p is the DCG at rank p (as shown above) and the $IDCG_p$ is the DCG value at rank 'p' for the ideal ranking. This ideal ranking is the ranking of these pairs in decreasing order of their relevance scores i.e just like we had in the above part. We get:

- nDCG at 50 : 0.3521042740324887
- nDCG at whole dataset : 0.5979226516897831

4. We extract the value of the feature 75 and then rank the query-URL pairs on the basis of that feature i.e- we sort the query-url pairs in the decreasing order of the value of feature 75 because it is given that the higher the value, the more relevant the URL. This sorting in decreasing order based on feature 75's value gives us the required ranking. Also, it is given that query-url pairs with non-zero relevance judgement scores are considered relevant. Thus using this we can get the total number of relevant pairs according to this technique. Now to get precision, recall values we calculate the value of precision@K and recall@K for k = 1 to 103 (total number of query url pairs with qid:4). We plot these values to get the precision-recall curve:



Question - 3

We needed the documents of these 5 classes for our task : ["comp.graphics", "sci.med", "talk.politics.misc", "rec.sport.hockey", "sci.space"]. Thus we pick the documents in their directories only. We have used the ‘os’ library of python to traverse the respective directories and pick files from the data directory.

Preprocessing:

After extracting the text from the files we performed the following preprocessing steps on the text to convert them into a list of tokens for each file.

- First, we converted the text to lowercase.
- Text tends to have the shortened form of a combination of two words For Example “I will” is written as “I’ll” to fix this issue we have fixed the contractions from the text. We expand the contractions for eg. convert “don’t” to “do not”
- We then performed word tokenization on the text.
- For all tokens, non-alphanumeric characters were removed from them.
- We then removed all the stopwords from our tokens.
- Punctuations and blank spaces were removed from the tokens and finally, we had our valid sets of tokens.
- Stemming is performed on the tokens

Methodology:

The data of each class is split into training and testing sets according to the given train:test split ratio. We shuffle the data samples before splitting to ensure random splitting. After this, we process the data to get the set of tokens for each sample. After this processing, we get three dictionaries:
-> *class_wise_tokens* - nested dictionary with the outer dictionary having the key as the class_label and the value as another dictionary that contains

the tokens for each preprocessed document in the training and testing set of that class, -> *class_wise_train_unique_tokens* - A dictionary with key as the class_label and value is a list of all the unique tokens present in the documents of that particular class, -> class_wise_train_tfs - A dictionary with key as the class_label and value is a dictionary that has the term frequencies of all the unique terms in that class. Using this then we calculate the ICF for all the unique tokens we have in all the documents in the training set. For ICF calculation, we use the approach of making a dictionary that has the keys as the unique terms in training set across all the classes and the value they point to is a list of class labels they occur in. This way, we can get its class frequency (CF) i.e number of classes in which that term occurs which would be the length of the list which that term points to in the dictionary. So using this approach, we calculate the CF for each term and thus the ICF for each term which is :

$$ICF_{term} = \log(N / CF_{term})$$, where N represents the number of classes

We then perform feature selection using the TF-ICF technique.

TF-ICF score for a term ‘t’ belonging to a class ‘c’ is equal to

$$TF\text{-}ICF_{t,\,c} = TF_{t,\,c} \times ICF_{t}$$

where $TF_{t,c}$ = No. of occurrences of a ‘t’ in all documents of class ‘c’

This $TF_{t,c}$ can be obtained from the dictionary ‘class_wise_train_tfs’ we have formed earlier. Note that the feature selection is performed over the training set. We thus obtain the TF-ICF scores for all the terms wrt a particular class and sort them in descending order and select the ‘k’ features(terms) corresponding to the top ‘k’ TF-ICF scores. This is done for all the classes and top ‘k’ features are selected for each class. Subsequently, our final feature set i.e the effective vocabulary is the union of the top k features of each class. Let the dimension(size) of the final vocabulary be |v|. Note that here features correspond to terms/words/tokens. Now we have our final feature set / effective

vocabulary using which we featurize the training and testing data to form |v| dimensional samples corresponding to the feature set we have and we get train_x, train_y, test_x, test_y using this process where train(/test)_x represents the samples(samples with feature values) in training data and train(/test)_y is the corresponding class label.

We train the Naive Bayes model using this data. Here we find the class prior probabilities for each class which is the ratio of the number of samples of that class to the total number of samples which is :

$$P(C_i) = (\text{no. of samples of class } C_i) \ / \ (\text{Total no. of samples})$$

We also calculate the class conditional probability which is the likelihood for each of the features wrt each class. For a feature x_i and class c_i this value is:

$$P(x_i \mid c_j) = N_{x\_i,\, c\_j} \ / \ \Sigma_k(N_{x\_k,\, c\_j}) \ \text{ where } k = 1 \ldots |v|$$

and $N_{t,\, c}$ no. of occurrences of a ‘t’ in all documents of class ‘c’. Also to avoid zero probabilities we have used laplacian smoothing here.

Using this, we find these likelihood probabilities for all the features wrt to all the classes. The Naive Bayes model is trained in this way. For testing, we calculate the posterior probability of each class given the sample and select the class corresponding to the maximum posterior probability. Using the Bayes theorem we have

$$P(C_j \mid X) = [P(X \mid C_j) * P(C_j)] \ / \ P(X)$$

The denominator here is effectively constant thus we only calculate the numerators and ignore the denominator.

Also, sample X is composed of features $[x_1, x_2 \ldots\ldots x_{|v|}]$ thus we have the formulation of the posterior probability as :

$$P(C_j \mid X) \propto [P(x_1 \mid C_j) * P(x_2 \mid C_j) * \ldots\ldots * P(x_{|v|} \mid C_j)\ ] * P(C_j)$$

also, we use log probabilities here as dealing with such low values of probabilities can cause underflow. Thus we effectively have this product mentioned above written as the sum of logs of the probabilities in the RHS. We choose the class with the maximum value of the posterior probability calculated (maximum proportional to posterior to be precise). This is the working of the Naive Bayes model.

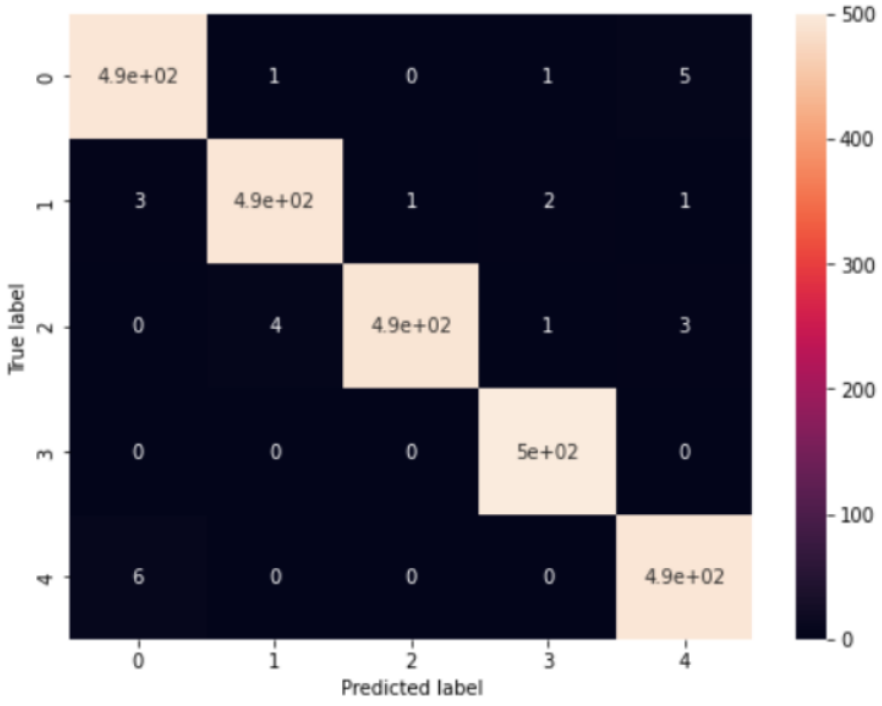
Performance Analysis:

First, we see the variation in the accuracy with respect to train:test split ratio size. For this analysis, we fix the value ok ‘k’ to be 30 i.e the no. of top-k features selected from each class to be 30. For this, the following are the statistics for different train: test split ratio :

1. **50:50**

Accuracy = 98.88%

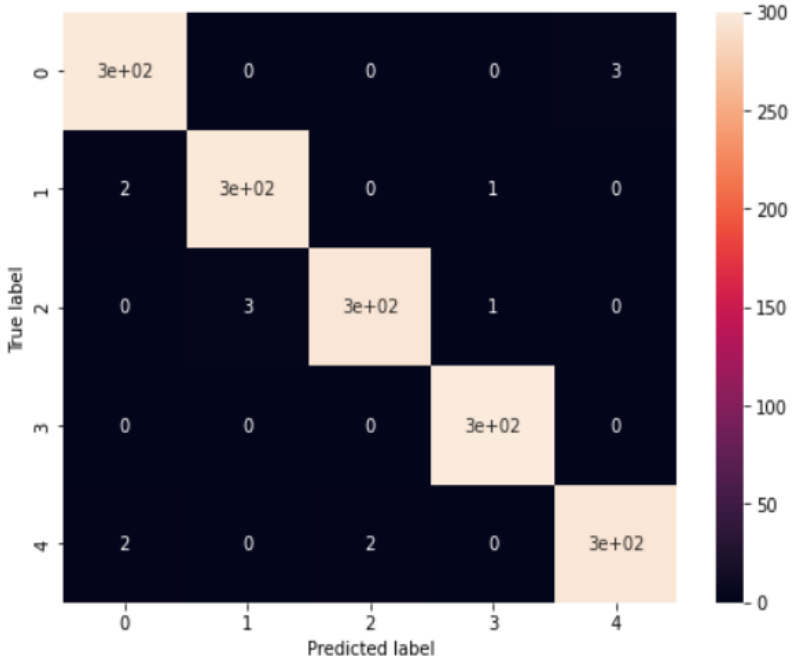
Confusion Matrix:
[[493. 1. 0. 1. 5.]
[3. 493. 1. 2. 1.]
[0. 4. 492. 1. 3.]
[0. 0. 0. 500. 0.]
[6. 0. 0. 0. 494.]]



2. **70:30**

Accuracy = 99.06666666666666%

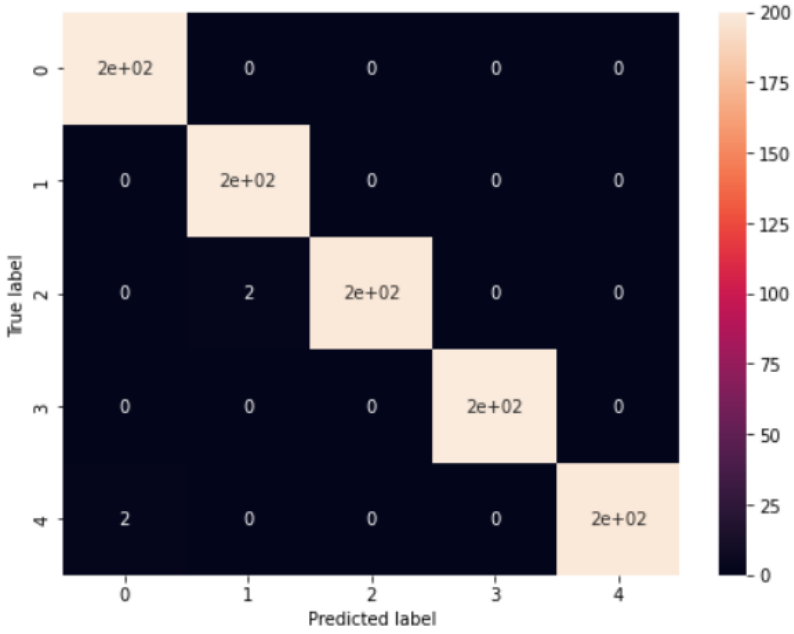
Confusion Matrix:
[[297. 0. 0. 0. 3.]
[2. 297. 0. 1. 0.]
[0. 3. 296. 1. 0.]
[0. 0. 0. 300. 0.]
[2. 0. 2. 0. 296.]]



3. **80:20**

Accuracy = 99.6%

Confusion Matrix:
[[200. 0. 0. 0. 0.]
[0. 200. 0. 0. 0.]
[0. 2. 198. 0. 0.]
[0. 0. 0. 200. 0.]
[2. 0. 0. 0. 198.]]



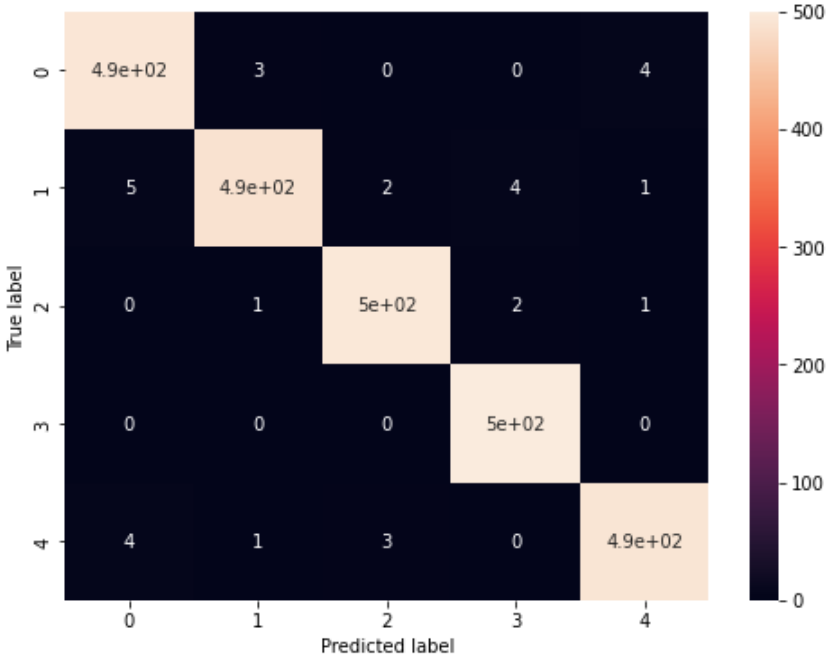
Thus, we can see that with the increase in the size of the training data set, the accuracy increases. There is a lower percentage of misclassified samples as is visible from the confusion matrix. The model when trained using more data, fits the data better i.e it models the pattern in the data in a

better way leading to a better generalization in turn. Thus, it will perform more accurate classifications.

Trying out for other values of k :

k = 50

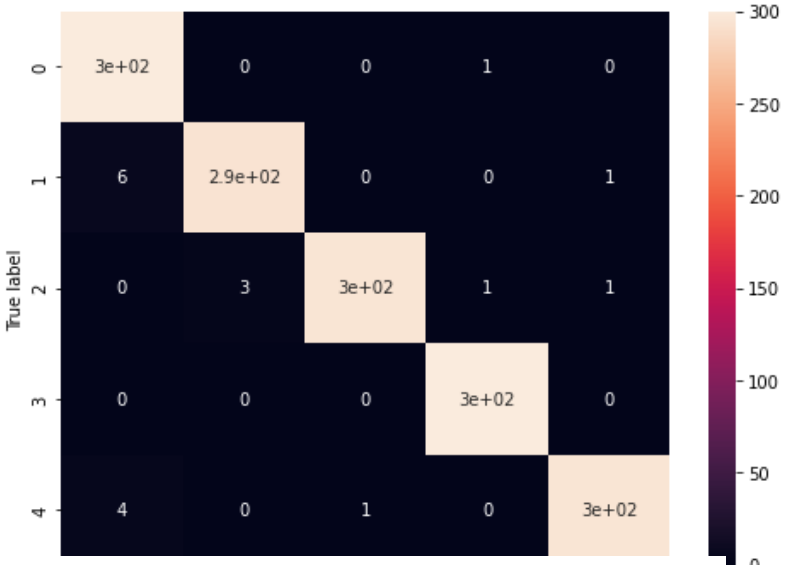
1. 50:50



Accuracy = 98.76%

Confusion Matrix:
[[493. 3. 0. 0. 4.]
[5. 488. 2. 4. 1.]
[0. 1. 496. 2. 1.]
[0. 0. 0. 500. 0.]
[4. 1. 3. 0. 492.]]

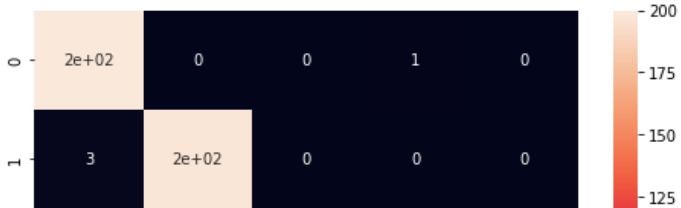
2. 70:30



Accuracy = 98.8%

Confusion Matrix:
[[299. 0. 0. 1. 0.]
[6. 293. 0. 0. 1.]
[0. 3. 295. 1. 1.]
[0. 0. 0. 300. 0.]
[4. 0. 1. 0. 295.]]

3. 80:20



Accuracy = 98.9%

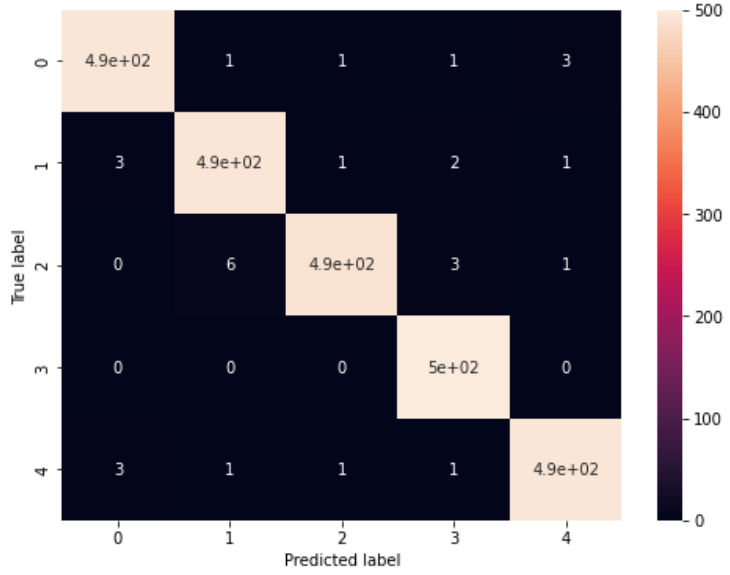
Confusion Matrix:
[[199. 0. 0. 1. 0.]
 [3. 197. 0. 0. 0.]
 [0. 2. 198. 0. 0.]
 [0. 0. 0. 200. 0.]
 [5. 0. 0. 0. 195.]]

k = 100 :

1. 50:50

Accuracy = 98.83999999999999%

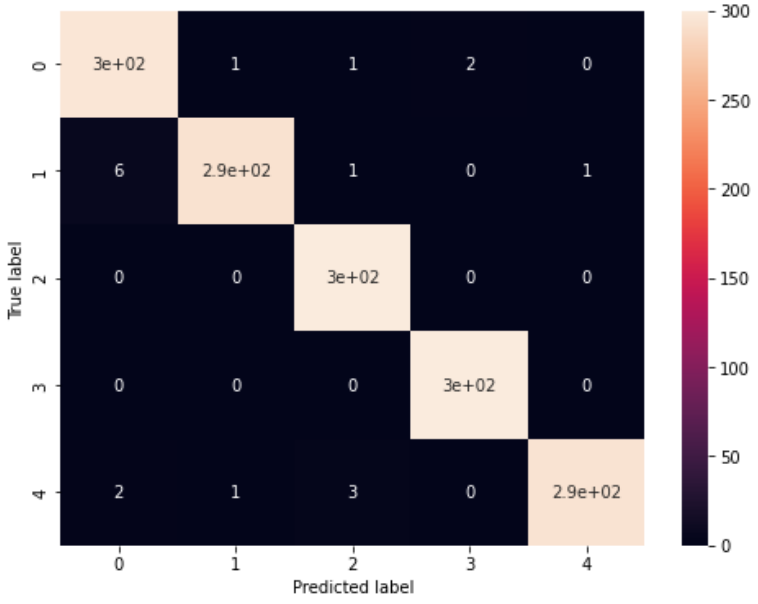
Confusion Matrix:
[[494. 1. 1. 1. 3.]
 [3. 493. 1. 2. 1.]
 [0. 6. 490. 3. 1.]
 [0. 0. 0. 500. 0.]
 [3. 1. 1. 1. 494.]]



2. 70:30

Accuracy = 98.8%

Confusion Matrix:
[[296. 1. 1. 2. 0.]
 [6. 292. 1. 0. 1.]
 [0. 0. 300. 0. 0.]
 [0. 0. 0. 300. 0.]

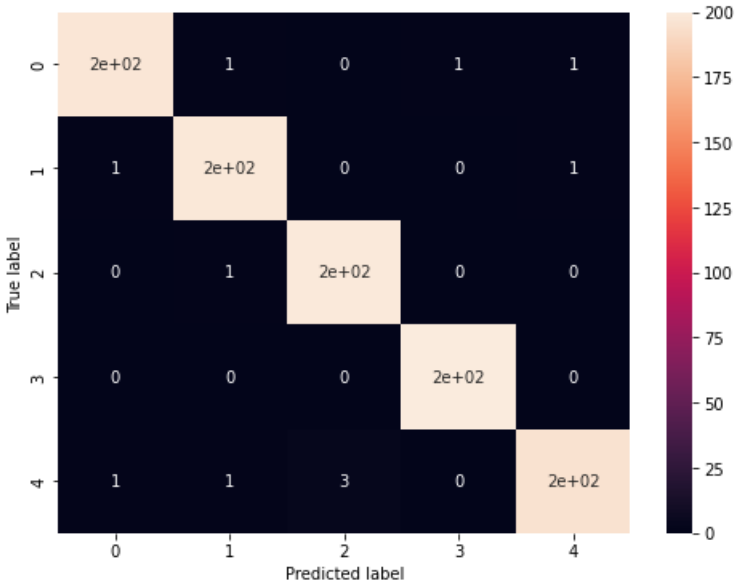


[2. 1. 3. 0. 294.]]

3. 80:20

Accuracy = 98.9%

Confusion Matrix:
[[197. 1. 0. 1. 1.]
[1. 198. 0. 0. 1.]
[0. 1. 199. 0. 0.]
[0. 0. 0. 200. 0.]
[1. 1. 3. 0. 195.]]



Thus, we can say that mostly the trend remains that with the increase in training size, the accuracy increases. However, sometimes this may not happen but that will be very sample split specific as in the case ok k=100 we see that accuracy does not vary much from 50:50 to 70:30 but it does increase if we increase the training size further to 80:20. With respect to variation in the value of ‘k’, we do not observe a very consistent/fixed trend wrt accuracy across different values train: test split. There is no significant improvement if we increase our ‘k’ i.e feature count very much which may mean that the most important features that capture the patterns are limited to some top features. Feature selection may or may not increase accuracy in general. It does significantly reduce the training time in the case of very large dimensional datasets. In some cases it can boost accuracy by removing noise from the dataset.