Report: First Part of the Assignment
Exploration with Feature Extraction, and Classical Data Analysis and Machine Learning Algorithms

Introduction

The first part of this project is about experimenting with the feature extraction methods, and the use of classical data analysis and machine learning algorithms on an image dataset. The image dataset was based on 'Fashion MNIST'. We were provided with training data of 60,000 samples with 786 features. We were also given the test data without their classes. Essentially, we experimented with this dataset for classification problems. In the following paragraphs, the details of the experiment and the answers to the questions of the assignment will be described. During our work we have consulted number of books, blogs, online resources as given in references [1] [2] [3] [4] [5] [6] [7] [8][9][10] [11][12][13][14] [15][16][17].

Answer to Question 1.1 Explanation of Design and Implementation Choices of your Model (Describe the underlying algorithms you are using, how they work and why you chose them)

Design Approach

Our approach for the first part of the project is listed down below:

- Learn about the Data.
- Explore Training Data set: checking class distribution.
- Explore Test Data set.
- Feature Engineering: Checking missing dataset.
- Splitting train and test data: We will do it on the Training dataset and rename our split test dataset as Validation dataset.
- Explore if the classes can be made categorical seeing the image pattern.
- Feature Scaling.
- Feature Extraction and visualization: Capturing maximum variance or representation of the features, we tried the followings:
    - PCA: Find out the best components range that captures most of the variance
    - Kernel PCA: For visualization
    - Locally-Linear Embedding (LLE): For visualization
    - t-distributed Stochastic Neighbor Embedding (t-SNE): For visualization
    - LDA: For visualization
- KNN:
    - Explore with hypermeters: n_neighbors = [3,5,11,19, 25, 29, 35, 45, 51], weights= ['uniform', 'distance'], metric= 'metric': ['euclidean', 'manhattan'].
    - Find the best parameters.
    - Fit the model with reduced dimension dataset.
    - Fit the model with all feature sets.
    - Check accuracy and other metrics.
    - Calculate the time for training and testing of the model.
- SVM:
    - Explore with hypermeters: kernel: ['linear', 'rbf'], C = [0.1, 0.5, 1, 5, 15, 35, 50], gamma = [1,0.1,0.01,0.001].
    - Find the best parameters.
    - Fit the model with reduced dimension dataset.
    - Fit the model with all feature sets.
    - Check accuracy and other metrics.

- Calculate the time for training and testing of the model.
- Use AdaBoost to check the result for enhancement.

- Decision Tree Classifier:
  - Explore with hypermeters: criterion= ['gini', 'entropy'], max_depth = [4, 6, 8, 12, None].
  - Find the best parameters.
  - Fit the model with all feature sets.
  - Check accuracy and other metrics.
  - Calculate the time for training the model.
  - Use AdaBoost to check the result for enhancements

- Random Forest Classifier:
  - Explore with hypermeters: kernel: n_estimators = [5, 10, 50, 150, 200, 300, 350, 650, 850], max_depth= [3, 5, 10, None].
  - Find the best parameters.
  - Fit the model with dimension reduced dimension dataset.
  - Fit the model with all feature sets.
  - Check accuracy and other metrics.
  - Calculate the time for training the model.
  - Use AdaBoost to check the result for enhancements

- Gradient Boosting Classifier: Use scikit-learn and XGBoost Library
  - Fit the model with all feature sets.
  - Check accuracy and other metrics.
  - Calculate the time for training and test of the model.

Learn about the Data

- **Data**: Two datasets are provided. One was the training data with 60,000 samples with associated classes. The second data set was a Test dataset with 10,000 samples with no classes associated with it.

- **Training Dataset**.

  - The training data consists of 60,000 samples with 786 features. Of 786 features, 784 features were pixel values of images. The rest two were information associated with the samples; one was ID and the other was classes. There is a total of five classes from 0 to 4. The classes could not be categorized as it has been observed that for a particular class the item in the image varied. For example, we found for class '3', there were images shoes and shirts. So, unlike the Fashion- MNIST dataset where the classes were categorized as 'shirt' or 'bag', the classes are just numeric numbers.

  - The training data contains 60,000 images of different clothing/ fashion items. Each image is 28 X 28 grayscale image intensity, instead of an RGB scale. So, the total pixel is 28X28 = 784 values, which are the features of the dataset. The pixel values are an integer between 0 and 255.
  - Out of 786 features, the first column was ID assigned to samples. The last feature was the classes associated with the sample.

- **Test Dataset**. The test dataset consists of 10,000 samples with 785 features. The first column is the ID of the samples. The rest 784 columns are the features in 784-pixel values. No class associated with the samples are given. So, this dataset was used to predict the model's accuracy and other metrics by submitting the predictions in Kaggle. The Kaggle showed the result in terms of accuracy and the score was posted to the leaderboard.

Explore the Training Data set

We first observed that the shape of the training dataset, which was (60000, 786). To see any distinction that we can make out of the data, we called the describe method. We noticed that almost all features had a maximum value of 255, except the first two features where the maximum value was 14 and 45. This made sense, as these were images of different clothing items, the corners of the image are lighter or white. So, some of the features in the datasets are likely to be of less intensity.

We needed to check the class distribution of the training dataset. We found that the five classes are almost equally distributed, around 120000 instances for each class (range between 11975 and 12067). This shows that the classes are nicely distributed. So, our training models are less likely to suffer from over or under-sampling.

```
[ ] df['Label'].value_counts()

    4    12067
    2    11994
    1    11986
    3    11978
    0    11975
    Name: Label, dtype: int64
```

```
[ ] sns.countplot(x = 'Label', data = df)

    <matplotlib.axes._subplots.AxesSubplot at 0x7f2ec85e1898>
```

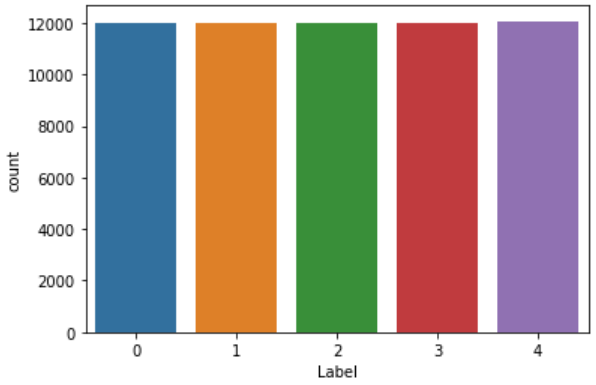


Figure: Class distribution in the dataset

Explore the Test Data set

We first observed the shape of the test dataset, which was (10000, 785). Since we do not have the classes associated with the data, we made a data frame with all features except the 'ID' for fitting in our models for prediction.

Feature Engineering

Feature engineering involves feature selection and feature extraction. Feature selection is about choosing the most useful features from all the features for training the model. Feature extraction is using exiting features to produce more useful features, such as using PCA, LDA or other manifold learning algorithms.

- **Feature Selection**. As our data were pixel values of a large-numbered and varied-shaped clothing images, so it was not feasible to try finding any particular feature for training. So, the options for the feature selection process was very limited in our dataset. But, one of the important parts of this data analysis was to see if there is any missing value. For any missing values in the data, we needed to conduct feature engineering before starting with our analysis. So, we checked for missing values. The training dataset had no missing value, which is nice. Because if we had any missing values, we might have to do some reasonable engineering to fill up the missing data.

```
[ ] df.isnull().sum()/len(df)

    Id        0.0
    Label     0.0
    1         0.0
    2         0.0
    3         0.0
              ...
    780       0.0
    781       0.0
    782       0.0
    783       0.0
    784       0.0
    Length: 786, dtype: float64

[ ] df.isnull().any().sum()

    0
```

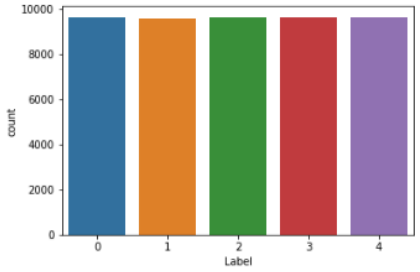
- **Feature Extraction**. It will be discussed in the later part of this report.

Splitting the Training and Test Data

For this project, we renamed the test dataset as Validation dataset to avoid confusion. We split the original training data into the train and validation dataset with 20% for testing. Thus, our training data for the analysis was 48000, 784 and validation data was 12000, 784 data, in addition to classes for train and test data.

For having a clear idea about how classes are distributed in our training data (48000, 784), we examined the distribution. The class distribution in the training data was also balanced, ranging between 9562 and 9630. So, our train is going to be balanced and it will not suffer from over or under-sampling.

```
[ ] X_train.shape, X_val.shape, y_train.shape, y_val.shape
((48000, 784), (12000, 784), (48000, 1), (12000, 1))
[ ] sns.countplot(x = 'Label', data = y_train)
<matplotlib.axes._subplots.AxesSubplot at 0x7f2ec3ed8438>
```




```
[ ] y_train['Label'].value_counts()
2    9630
3    9611
4    9601
0    9596
1    9562
Name: Label, dtype: int64
```

Feature scaling

Feature scaling is important when the features have a very large difference in scale. Our data sample is a black and white image that is a matric of pixels- an integer value ranging between 0 and 255. Apparently, for our task, the scale is defined i.e. integer values between 0 and 255. But, scaling can be useful for our analysis for the following reasons:

- Some of the classifiers are more sensitive to the scaling of features, like SVM. Some others are less affected by scaling, such as decision tree classifiers. So, we experimented with both scaled and unscaled data for different algorithms.
- Scaled data take less time for fitting the model for training that of unscaled data. Given a large dataset, scaling can be very useful.

Of the two popular scaling approaches, min-max and standard scaling, each has advantages and disadvantages. Min-max scaling transform the data into values between 0 and 1. On the other hand, standard scaling does not bound the range between any range. Standard scaled data are less affected by outliers. However, our data are less likely to have outliers as they are ranged between 0 and 255. Thus, we preferred to use the min-max normalization. However, we explored standard scaling for a few cases to check if our analysis was correct.

Feature Extraction

Dimensional reduction causes some loss of information, but it has two major benefits:

- It can speed up training in case of large data.
- It can help in visualizing class distribution by reducing the number of dimensions to only two dimensions.

Visualization

- **PCA for Visualization**. We used to see a two-dimensional projection of the data using the first two principal components of the data. We plotted the components according to classes. But, we found that the classes were not separable.

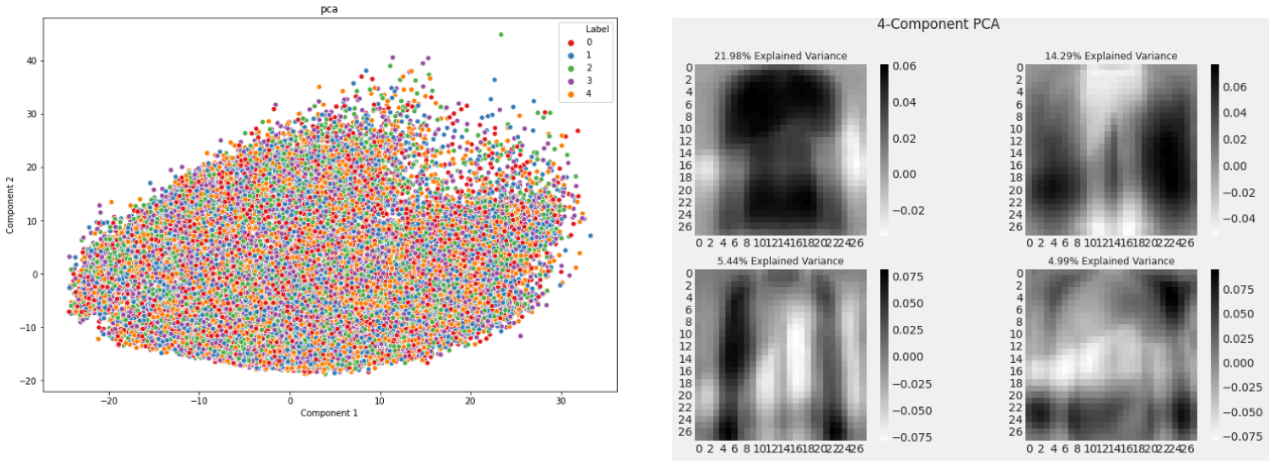


Figure: PCA projection

- **tSNE for Visualization**. t-SNE is sensitive to local structure and converts the affinities of the data points into probabilities. We used to see a two-dimensional projection of the data using the first two principal components of the data. We plotted the components according to classes. But, we found that the classes were not separable.

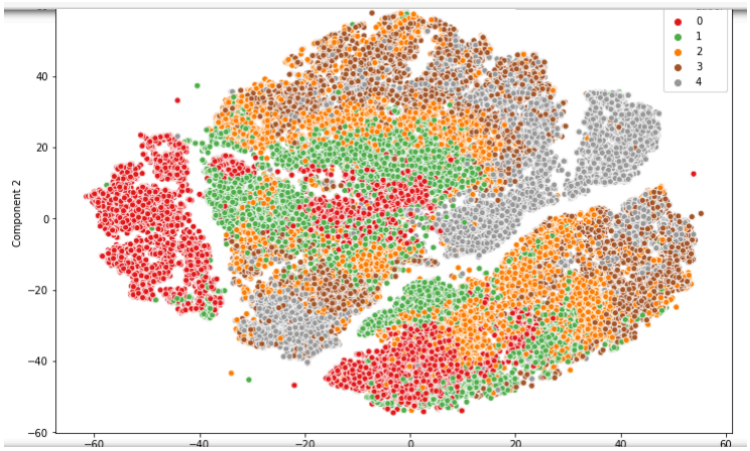


Figure: tSNE projection.

- **Locally Linear Embedding (LLE) for Visualization**. LLE builds a graph and represents the data in lower dimension. LLE projects the data by maintaining distances within the local neighbors. We found it to worst in case of class visualization as all data points were on the same place.

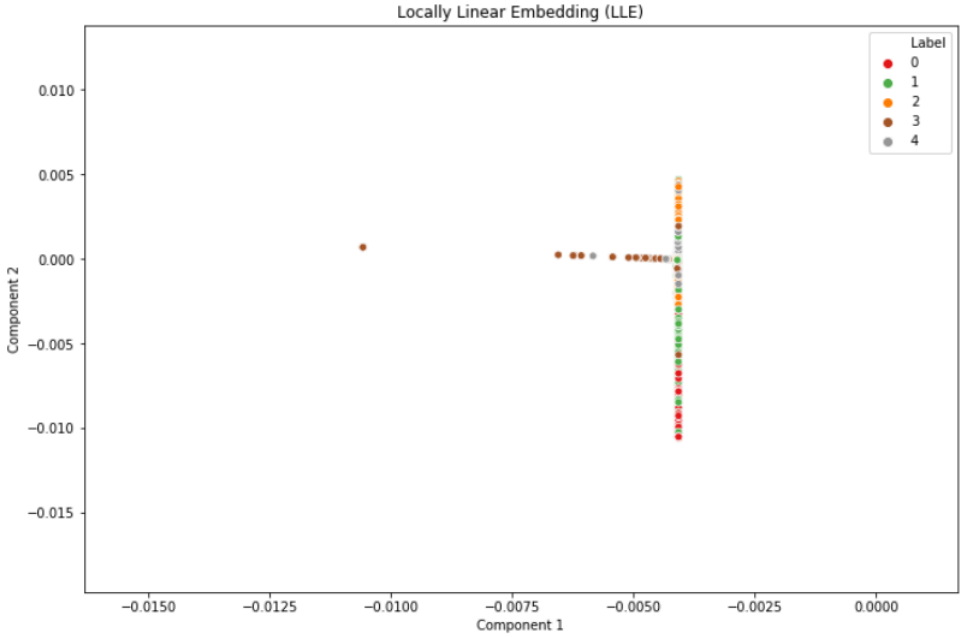


Figure: LLE projection.

- **LDA for Visualization**. Theoretically, we know that in addition to maximizing the variance of the data, LDA maximizes the class-spread. However, it also depends on the spread of the classes in the dataset. Given our data set, LDA could not better project the class segmentation.

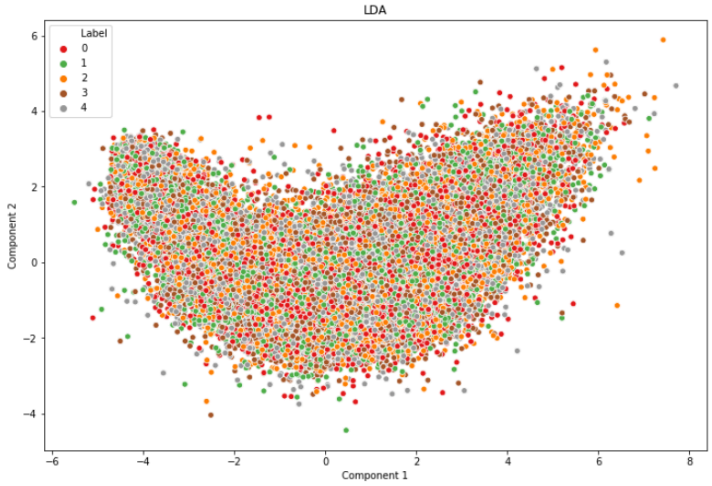


Figure: LDA projection.

Feature Extraction: Dimensionality Reduction

- **PCA for Training and Testing**. PCA accounts for preserving the variance in the data. Since our dataset is high-dimensional, we tried to find out the best collection of PCAs that capture the maximum variance of the data. As we used the scikit-learns library, our PCA models centered the data around the origin. We did it by carrying out test and trail approach.

    - **PCA of 784 Components**. We We first tried to find out all 784 principal components and understand how much variances are explained by the components. We found that that there was a sharp fall in the variance captured by the components after around 150 components. Again, it is seen that around 100 components capture more than 85% of the variance of the dataset. But, we

need a clear plotting of the principal components. From the scree plot, we could see there was an indistinguishable variance after around 380 components. So, we then tried to see the contribution of the 380 principal components.

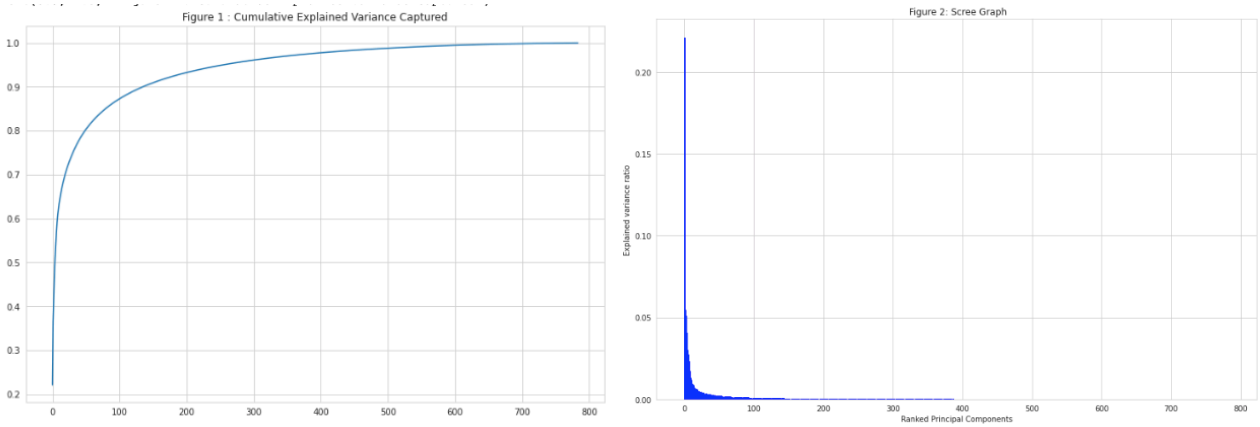


Figure: Principal components for all 784 features: Cumulative Explained Variance and Scree Plot.

- **PCA of 380 Components**. After we concentrated on 380 principal components, we found that more than 90% variance is captured by PCA around 135 components. From the scree plot, we also noticed that after 140 components the explained variance reduced sharply.

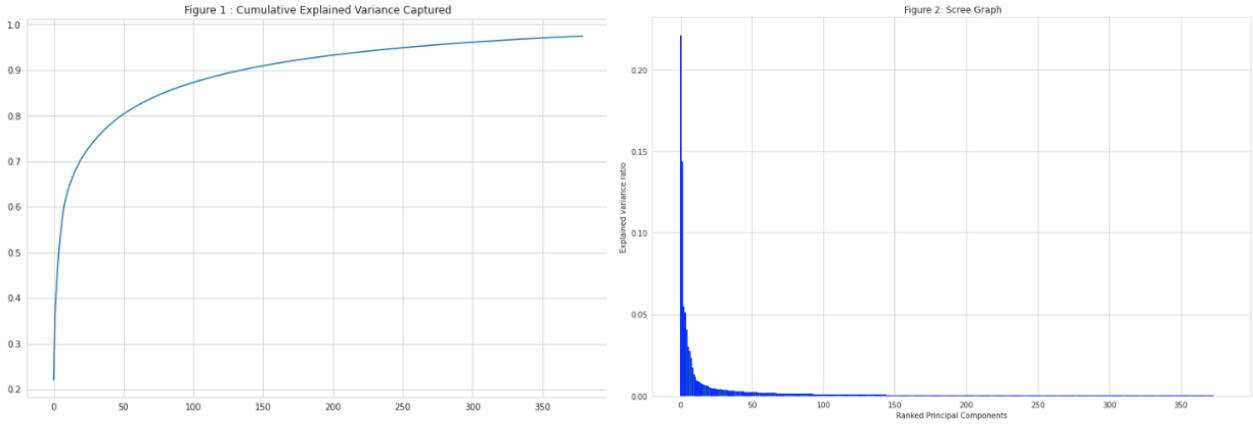


Figure: Principal components for all 380 features: Cumulative Explained Variance and Scree Plot.

- **PCA of 140 Components**.

  - From our analysis with 140 components, we noticed that 140 components can capture more than 90% variance of the data. We also noticed that after 140 components the remaining variance is very less. So, we print out the result of the variance captured by the components.

  - We found that only the first 11 components can capture variance of more than 1% and the rest of the component captures the rest 1% of the variance. This was so interesting that we tried to plot the first four components to construct the images which explained 21.98%, 14.29%, 5.44%, and 4.99% variance respectively. We found images that insightful. For the first image, we see the dark spots shaped like a ‘shirt’ image, while

the lighter spots were shaped like a ‘shoe’. For the second component, we see the exact opposite, darker were shoe shaped and lighter were shirt shaped. That was expectable and explain how PCA components are orthogonal to each other.

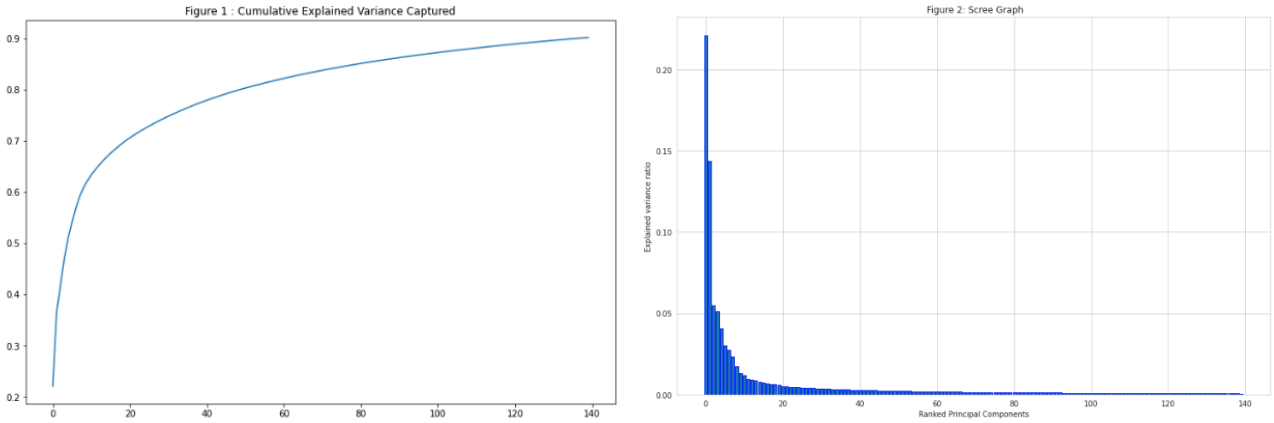


Figure: Principal components for all 140 features: Cumulative Explained Variance and Scree Plot.

```
22.11% variance is explained by 1 principal component
14.37% variance is explained by 2 principal component
5.47% variance is explained by 3 principal component
5.10% variance is explained by 4 principal component
4.07% variance is explained by 5 principal component
3.01% variance is explained by 6 principal component
2.75% variance is explained by 7 principal component
2.32% variance is explained by 8 principal component
1.71% variance is explained by 9 principal component
1.33% variance is explained by 10 principal component
1.17% variance is explained by 11 principal component
0.96% variance is explained by 12 principal component
0.90% variance is explained by 13 principal component
0.86% variance is explained by 14 principal component
0.76% variance is explained by 15 principal component
0.73% variance is explained by 16 principal component
0.67% variance is explained by 17 principal component
0.63% variance is explained by 18 principal component
0.62% variance is explained by 19 principal component
0.59% variance is explained by 20 principal component
```

Figure: Variance captured by the principal components

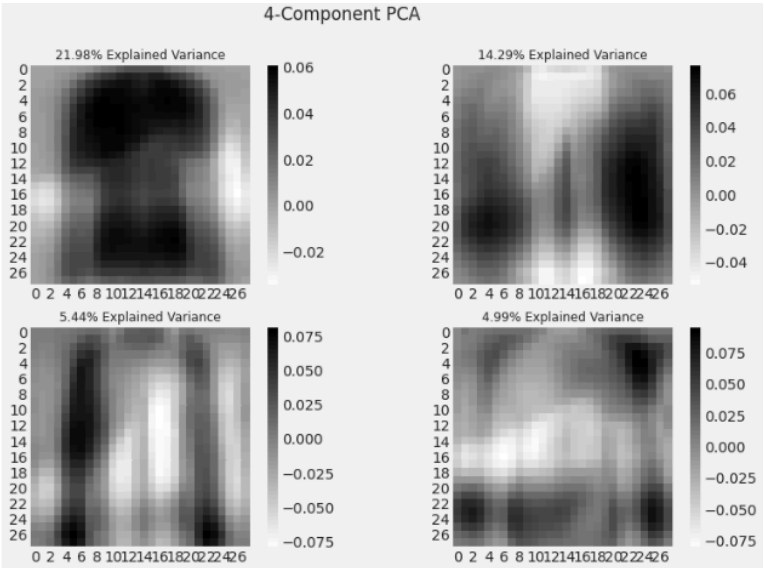


Figure: First 4 Principal Components

Finally, using the dimensionality reduction, we captured more than 90% of variance of the data set. Thus, this compression will reduce the training time for classifiers, especially for SVM.

Classification Algorithms Used

K-Nearest Neighbors (KNN)

We know that KNN is non-parametric, meaning that the number of parameters grows with the amount of training data and it can be completely intractable for large datasets. It is also a non-probabilistic classifier, i.e. it does not do any probability calculation on any data point for classification. Rather, the algorithm works on a distance function. It makes a prediction on test data by considering the nearest neighbors' classes from the training data and voting the weights for the classes.

The The important hyperparameters of KNN are n_neightbors, weights and metri. n_neighbors is the declaration of how many data points will be considered as neighbors for calculation of the distance function. Weights are also important as it provides the basis for assigning weights to the neighborhood votes. Finally, the distance metric that indicates how the distance will be calculated. To find the best hyperparameters, we tested the performance with the following combination (108 fits):

- n_neighbors = [3, 5, 11, 19, 25, 29, 35, 45, 51]: Using few neighbors makes the model complex while using a large number of neighbor make the model less complex. For example, taking only 1 neighbor will make the model very complicated as it calculates distance on the feature dimension of every data point. Again, taking a large number of neighbors simplifies the model and it will predict all test points to the same class that is more in the training data

- weights= ['uniform', 'distance']: For the voting weights, we tried both ‘uniform’ and ‘distance’. In uniform weight, regardless of the distance between the training data point and the test data point, the voting weight is the same. Thus, it does not provide any voting weight. But, in the case of ‘distance’ weight, the voting weight is affected by the distance of the training data point.

- metric= ['euclidean', 'manhattan']: While the most popular distance metric is Euclidean or L2 – a distance that just calculates the distance following the Pythagorean theorem, we also tested ‘Manhattan’ distance or L1- distance. It made more sense as the straight line distance measure provided better result than the block-based distance. The straight line distance measure provided better result than the block-based distance

```
[ ] grid_params = {'n_neighbors' : [3,5,11,19, 25, 29, 35, 45, 51],
                   'weights': ['uniform', 'distance'],
                   'metric': ['euclidean', 'manhattan']}

[ ] model_tuner = GridSearchCV(KNeighborsClassifier(),grid_params,cv=3, refit=True,verbose=3)
    model_tuner.fit(X_pca140,y_train)
    model_tuner.best_params_,  model_tuner.best_estimator_, model_tuner.best_score_
```

Figure: Hyperparameters for KNN

```
  self.best_estimator_.fit(X, y, **fit_params)
({'metric': 'euclidean', 'n_neighbors': 11, 'weights': 'distance'},
 KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='euclidean',
                      metric_params=None, n_jobs=None, n_neighbors=11, p=2,
                      weights='distance'),
 0.8439166666666668)
```

Figure: Best parameter from GridSearchCV on Hyperparameters for KNN

One of the limitations of KNN is that it performs badly with high dimensional spaces. As our data had high dimensions, feature extraction provided better results for our model. KNN is sensitive to data preprocessing. KNN is slow when the number of features is more. Again, KNN is biased towards classes with more samples.

The first reason is to explore and find how KNN does with our dataset. KNN is one of the simplest classifiers that provide better predication.

Since our class distribution as balanced, so the limitation of KNN that suffers from bias towards with more samples is not applicable.

We found that KNN was sensitive to scaling. It requires preprocessing of the data in the form of scaling and feature reduction. We used MinMaxScaler from scikit-learn. We found a significant improvement in the accuracy score from 71% to 85% after using the scaling. Scaled data also took less time for testing.

```
Time for KNN: 7.714900004884839e-05
[[2153  211   13    2    0]
 [ 807 1277  268   68    4]
 [ 392  309 1407  217   39]
 [   7   76  456 1717  111]
 [  33   65   71  363 1934]]

              precision    recall  f1-score   support

           0       0.63      0.91      0.75      2379
           1       0.66      0.53      0.59      2424
           2       0.64      0.60      0.61      2364
           3       0.73      0.73      0.73      2367
           4       0.93      0.78      0.85      2466

    accuracy                           0.71     12000
   macro avg       0.72      0.71      0.70     12000
weighted avg       0.72      0.71      0.70     12000
```

```
Time for KNN: 0.00012208900079713203
[[2280   92    6    1    0]
 [ 202 2005  162   50    5]
 [ 102  251 1763  205   43]
 [   2   41  241 1935  148]
 [  18   10   32  203 2203]]

              precision    recall  f1-score   support

           0       0.88      0.96      0.92      2379
           1       0.84      0.83      0.83      2424
           2       0.80      0.75      0.77      2364
           3       0.81      0.82      0.81      2367
           4       0.92      0.89      0.91      2466

    accuracy                           0.85     12000
   macro avg       0.85      0.85      0.85     12000
weighted avg       0.85      0.85      0.85     12000
```

Figure: Accuracy and time before and after scaling of data for KNN.

But, since our data had high-dimension, it was computationally expensive for testing.

Support Vector Machine (SVM)

SVM is a discriminative classifier that constructs a hyperplane in a high dimensional space that maximizes the distance to the nearest data point in each class and can be used for classification. It is also called a large margin classification. So, with training data, the algorithm can produce a hyperplane that can be used for classifying a new test data point. It does this by transforming the data points into a higher-dimensional space. This transform is known as 'kernel'. There are a number of kernel functions that can be used for SVM. In scikit-learn, there are four choices: 'linear', 'polynomial', 'rbf' and 'sigmoid'. However, we can also provide custom kernel functions for the transformation.

SVM is generally a binary classifier. So, we had to use strategies to utilize this binary classifier for running multiple binary classifiers. The strategies commonly used are one-versus-all and one-versus-one. For one-versus-all, we select the highest score of the classifier that attains the best result. For our cases, we have 5 classes, so each class scores are compared with the scores with the rest of the classes. In Scikit-Learn, LinearSVC uses this strategy. The runtime for One-vs-all is less. However, for our analysis, we used SVC that uses a one-versus-one strategy. So, in our case, (5* (5-1)/2) = 10 binary classifiers were trained that demanded a large amount of time. Since SVM scales poorly with the training set, the one-versus-one strategy is generally preferred.

SVM is sensitive to scaling. It works better with scaling between 0 and 1. So, we used MinMaxScaler from scikit-learn. SVM is generally recommended for a small dataset, generally less than 10,000 samples. Because it

demands more runtime and memory for larger datasets. But, as our dataset were pixel intensities and were on the same scale between 0 and 255, it was worth trying this classifier.

The important hyperparameters of SVM are the regularization parameter (C parameter), kernel function, and gamma parameter.

- Regulation Parameter (C parameter): C parameter deals with the misclassification of the training data points. It dictates the soft margin classification. Larger C value provides the smaller-margin hyperplane, but accept less margin violation, and lower C value provides the larger-margin for hyperplane but accepts more margin violation. Thus, higher C value allows each data point to have greater influence in the model for classification boundary. For our model, we used seven C value [0.1, 0.5, 1, 5, 15, 35, 50].

- Kernel Parameter: We tried with two kernels, 'linear' and 'rbf'. The linear model is generally better when the number of features is large compared to the training samples. But, as our training samples are large, the linear kernel took a significant amount of time. Since our dataset is nonlinear, it made sense to use 'rbf' which is defined by the Gaussian Radial Basis Function. One of the disadvantages of the rbf kernel is that it transformed our 60000 samples and 784 features into a training set of 60000 samples and 60000 features.

- Gamma: For SVM with 'rbf', gamma is an important hyperparameter. The hyperparameter gamma influences the shape/ width of the gaussian curve. With high gamma value, the curve becomes narrower and thus the decision boundary adjusts with each class data point. With smaller gamma value, the shape of the curve becomes wider and the decision boundary becomes smoother.

```
# Gamma value for scale
sca = 1/(len(X_pca140)* X_pca140.var())
print ('The scale value for Gamma:' , sca)

#Gamma value for auto
au = 1/len(X_pca140)
print ('The auto value for Gamma: ', au)
```

```
The scale value for Gamma: 4.1229571708831695e-06
The auto value for Gamma:  2.0833333333333333e-05
```

```
param_grid1 = {'C': [0.1, 0.5, 1, 5, 15], 'kernel': ['rbf'], 'gamma': [1,0.1,0.01,0.001]}
```

```
grid_svm1 = GridSearchCV(SVC(random_state=42),param_grid1, refit=True,verbose=3)
grid_svm1.fit (X_pca140, np.ravel(y_train))
```

Figure: Experiment with gamma value.

Since SVM take a long time for training, we first tried with 55 PCA with linear and rbf kernel. We found only 56% of accuracy for linear kernel. But, for rbf kernel, we found a definitive improvement to 74% for 55 principal components.

```
Time for SVM: 863.5133038780004
[[1488  817   73    1    0]
 [ 941  851  563   44   25]
 [ 330  422 1113  380  119]
 [  17   78  364 1514  394]
 [  50   82   72  547 1715]]

              precision    recall  f1-score   support

           0       0.53      0.63      0.57      2379
           1       0.38      0.35      0.36      2424
           2       0.51      0.47      0.49      2364
           3       0.61      0.64      0.62      2367
           4       0.76      0.70      0.73      2466

    accuracy                           0.56     12000
   macro avg       0.56      0.56      0.56     12000
weighted avg       0.56      0.56      0.56     12000
```

```
Time for SVM: 77.17771046000053
[[1965  317   95    0    2]
 [ 545 1543  284   38   14]
 [ 264  335 1459  256   50]
 [   1   66  344 1768  188]
 [  12   33   35  227 2159]]

              precision    recall  f1-score   support

           0       0.71      0.83      0.76      2379
           1       0.67      0.64      0.65      2424
           2       0.66      0.62      0.64      2364
           3       0.77      0.75      0.76      2367
           4       0.89      0.88      0.89      2466

    accuracy                           0.74     12000
   macro avg       0.74      0.74      0.74     12000
weighted avg       0.74      0.74      0.74     12000
```

Figure: Accuracy for linear and rbf kernel for 55 principal components.

```
GridSearchCV(cv=10, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=42, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.1, 0.5, 1, 5, 15], 'kernel': ['rbf']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=2)
```

```
[ ] grid_svm.best_params_,  grid_svm.best_estimator_, grid_svm.best_score_
```

```
({'C': 15, 'kernel': 'rbf'},
 SVC(C=15, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
     decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
     max_iter=-1, probability=False, random_state=42, shrinking=True, tol=0.001,
     verbose=False),
 0.8875208333333333)
```

Figure: Experiment with hyperparameters of SVM.

```
grid_svm.param_grid['C'], grid_svm.cv_results_['mean_test_score']
```

```
([0.1, 0.5, 1, 5, 15],
 array([0.8255    , 0.85970833, 0.870875  , 0.88477083, 0.88752083]))
```

```
Gridsearch scores on C value:

0.826 (+/-0.013) for {'C': 0.1, 'kernel': 'rbf'}
0.860 (+/-0.013) for {'C': 0.5, 'kernel': 'rbf'}
0.871 (+/-0.013) for {'C': 1, 'kernel': 'rbf'}
0.885 (+/-0.008) for {'C': 5, 'kernel': 'rbf'}
0.888 (+/-0.009) for {'C': 15, 'kernel': 'rbf'}
```

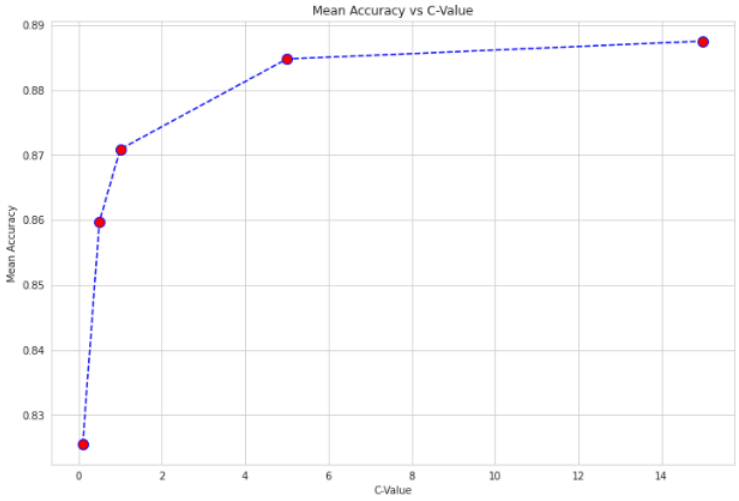
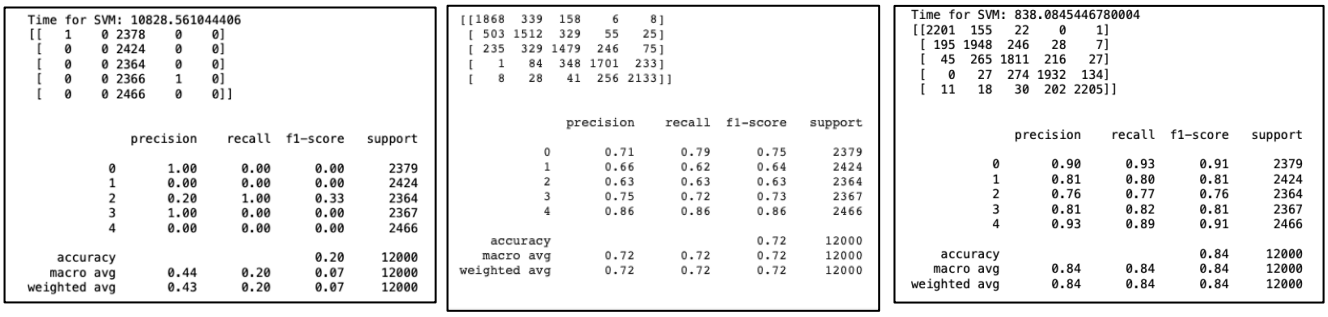


Figure: Effect on accuracy for C-value.

SVM is sensitive to the scaling of the data. It requires preprocessing of the data in the form of scaling and feature reduction. It works better with scaling between 0 and 1. We found a stark difference in accuracy, where without scaling, all test data were predicted to be in the same class.



```
Time for SVM: 10828.561044406
[[   1    0 2378    0    0]
 [   0    0 2424    0    0]
 [   0    0 2364    0    0]
 [   0    0 2366    1    0]
 [   0    0 2466    0    0]]

              precision    recall  f1-score   support

           0       1.00      0.00      0.00      2379
           1       0.00      0.00      0.00      2424
           2       0.20      1.00      0.33      2364
           3       1.00      0.00      0.00      2367
           4       0.00      0.00      0.00      2466

    accuracy                           0.20     12000
   macro avg       0.44      0.20      0.07     12000
weighted avg       0.43      0.20      0.07     12000
```

```
[[1868  339  158    6    8]
 [ 503 1512  329   55   25]
 [ 235  329 1479  246   75]
 [   1   84  348 1701  233]
 [   8   28   41  256 2133]]

              precision    recall  f1-score   support

           0       0.71      0.79      0.75      2379
           1       0.66      0.62      0.64      2424
           2       0.63      0.63      0.63      2364
           3       0.75      0.72      0.73      2367
           4       0.86      0.86      0.86      2466

    accuracy                           0.72     12000
   macro avg       0.72      0.72      0.72     12000
weighted avg       0.72      0.72      0.72     12000
```

```
Time for SVM: 838.0845446780004
[[2201  155   22    0    1]
 [ 195 1948  246   28    7]
 [  45  265 1811  216   27]
 [   0   27  274 1932  134]
 [  11   18   30  202 2205]]

              precision    recall  f1-score   support

           0       0.90      0.93      0.91      2379
           1       0.81      0.80      0.81      2424
           2       0.76      0.77      0.76      2364
           3       0.81      0.82      0.81      2367
           4       0.93      0.89      0.91      2466

    accuracy                           0.84     12000
   macro avg       0.84      0.84      0.84     12000
weighted avg       0.84      0.84      0.84     12000
```

Figure: SVM without any scaling, SVM with StandardScaler scaling (140 PC), SVM with MinMaxScaler

Besides, the runtime performance of the SVM is also dependent on the dimension of the feature vectors.

Decision Tree Classifier

The Decision Tree Classifier is a non-parametric model that tends to closely fit with training data. The learning of the classifier means it retains the sequence of if/else questions or test points. One of the many advantages of the Decision Tree is that it requires very little data preparation. In particular, it does not require feature scaling or centering at all. Since each node only requires checking the value of one feature, the overall prediction complexity is just O(log(m)), independent of the number of features. So predictions are very fast, even when dealing with a large training dataset.

We used scikit-learn, so it used the CART (Classification and Regression Tree) algorithm that produces only binary trees. So, it was pure yes and no. It is based on a greedy algorithm that makes choices in each split taking the best solution, but it is not an optimal solution. As the optimal solution is an NP-complete problem, that has a runtime of O(exp(n)), this approach is quite faster, runtime of $O(\log_2 n)$ – just as like a binary tree algorithm.

The overarching advantage of a Decision tree classifier is that it is easy to explain as it depends on simple classification rules. However, on the downside, it is prone to overfitting. The probability of overfitting increases as the tree gets deeper.

Since post-pruning is not available in scikit-learn, so we tried to pre-pruning by limiting the maximum depth. The important regularization parameter for the decision tree classifier is the max_depth. This hyperparameter limits the CART algorithm to stop splitting the training set. So, it acted as a stopping criterion for the running of our model. This is also essential for limiting the overfitting of the training data. We uses a combination of max_depth = [4, 6, 8, 12, None].

Another important parameter of the decision tree classifier is the criterion for the information impurity. In scikit-learn, we had two choices for information integrity or for reducing the impurity, gini and entropy. For example, a node is pure if its gini is 0. However, they both often result in a similar tree. Since our classes were well distributed, gini was a better choice. It runs faster than entropy. However, entropy would be a good choice if our data classes were not balanced, as it leads to a more balanced tree. We run on both criteria.

```
h_param_dtc = {'criterion':['gini', 'entropy'], 'max_depth':[4, 6, 8, 12, None]}
```

```
grid_dtc = GridSearchCV(DecisionTreeClassifier(random_state=42),h_param_dtc, refit=True,verbose=3)
grid_dtc.fit (X_train, np.ravel(y_train))
```

```
GridSearchCV(cv='warn', error_score='raise-deprecating',
             estimator=DecisionTreeClassifier(class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              presort=False, random_state=42,
                                              splitter='best'),
             iid='warn', n_jobs=None,
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': [4, 6, 8, 12, None]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=3)
```

```
grid_dtc.best_params_
```

```
{'criterion': 'gini', 'max_depth': 12}
```

Figure: Decision Tree Classifier hyperparameter experiment

With Decision tree classifier best parameters, we got a poor accuracy.

```
The training time for Random Forest Classifier :  20.755627729000025
[[2025  293   53    2    6]
 [ 297 1655  403   49   20]
 [  76  429 1409  390   60]
 [   6   75  384 1638  264]
 [  16   49  105  395 1901]]


              precision    recall  f1-score   support

           0       0.84      0.85      0.84      2379
           1       0.66      0.68      0.67      2424
           2       0.60      0.60      0.60      2364
           3       0.66      0.69      0.68      2367
           4       0.84      0.77      0.81      2466

    accuracy                           0.72     12000
   macro avg       0.72      0.72      0.72     12000
weighted avg       0.72      0.72      0.72     12000
```

```
DTC with AdaBoost time for training is 897.0444833190013 and time for test is 1.5386785619994043


Accuracy: 80.39%


[[2109  236   33    0    1]
 [ 119 1935  335   32    3]
 [  35  257 1808  235   29]
 [   0   37  385 1804  141]
 [  12   37   45  381 1991]]


              precision    recall  f1-score   support

           0       0.93      0.89      0.91      2379
           1       0.77      0.80      0.79      2424
           2       0.69      0.76      0.73      2364
           3       0.74      0.76      0.75      2367
           4       0.92      0.81      0.86      2466

    accuracy                           0.80     12000
   macro avg       0.81      0.80      0.81     12000
weighted avg       0.81      0.80      0.81     12000
```

```
abc_dtc
```

```
AdaBoostClassifier(algorithm='SAMME.R',
                   base_estimator=DecisionTreeClassifier(class_weight=None,
                                                         criterion='gini',
                                                         max_depth=12,
                                                         max_features=None,
                                                         max_leaf_nodes=None,
                                                         min_impurity_decrease=0.0,
                                                         min_impurity_split=None,
                                                         min_samples_leaf=1,
                                                         min_samples_split=2,
                                                         min_weight_fraction_leaf=0.0,
                                                         presort=False,
                                                         random_state=42,
                                                         splitter='best'),
                   learning_rate=1, n_estimators=50, random_state=None)
```

Figure: Decision Tree Classifier accuracy on classification. It improved when used with AdaBoost.

We tried to see if the prediction accuracy improves with boosting. We used AdaBoosting. As expected, the prediction accuracy improved from 72% to 80%. Details on AdaBoosting is given later on this report.

To get a visualization of the decision tree, we test with max_depth =4. Though we got a poor result, we could see how the tree was forming. To accommodate in the report, we have divided the tree graph into two (refer to Figure).

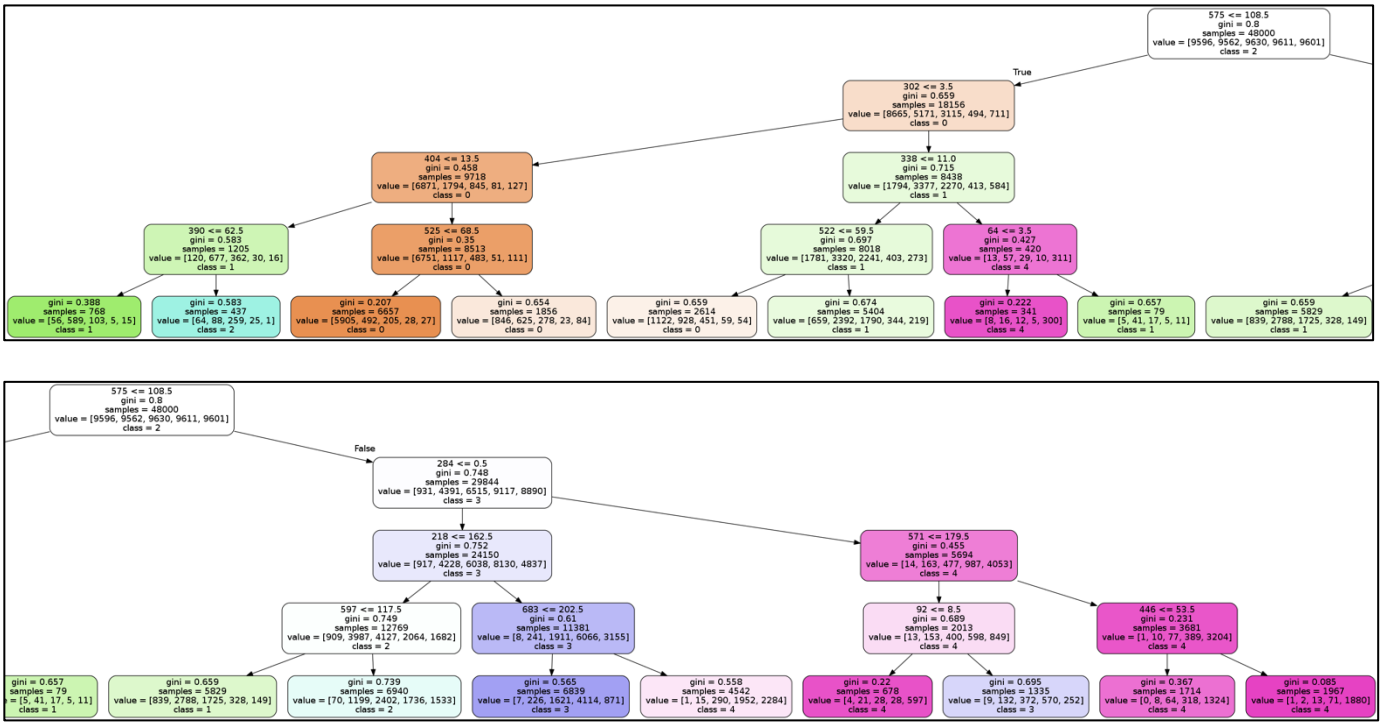


Figure: Decision Tree graph with maximum depth = 4. The graph is divided into two to accommodate in the report.

Random Forest Classifier

Random Forest classifier is an ensemble method and is a multi-class classifier. It performs well when the individual classifiers are independent of each other, thus not making the same error and improving the accuracy of classification. It uses the same algorithm as a decision tree classifier, but, it trains the models with different subsets of the training samples. This is often done with a replacement which is known as bootstrap aggregating. So, a random forest classifier combines the bagging classifier and decision tree classifier. It uses soft-voting for classification, i.e. it provides more weights on the best probability of the classifiers.

The problem of overfitting in Decision tree classifiers can be addressed by the Random Forest classifier by building multiple decision trees and vote them to classify. Thus, it avoids the overfitting by the decision tree by averaging the decision tree classifiers' results. It does by building random decision trees. Another advantage of the Random forest is that it captures a broader picture of the data than any other single tree. It does not require any scaling of the data.

However, the Random Forest classifier models are difficult to interpret, that is to say, that it is a sort of black-box model like neural network models. Because it builds a large number of trees, it is difficult to visualize and explain.

Hyper parameters:

- One of the important parameters of the random forest classifier is the max_feature. For our case, we used auto, i.e. n_features. So, in each split, it looked at all the features while maintaining the bootstrap property. Lower max_feature builds more diverse trees and higher max_features build more similar trees.

- 'n_estimators': [5, 10, 50, 150, 200, 650, 850],. The larger is always better as averaging more trees provide better probability and reduce the overfitting. But, a higher number of trees require more memory that we experienced while running, taking almost all the storage of the computer. It also took more time.
- 'max_depth': [3, 5, 10, None] }
- Bootstrap = true, so the BootStrap aggregating property maintained.
- Criterion = gini for decision tree information purity.
- Random State: Since Random Forest is highly random, it is essential to fix the random state to reproduce the result of the model. We set it to 42.

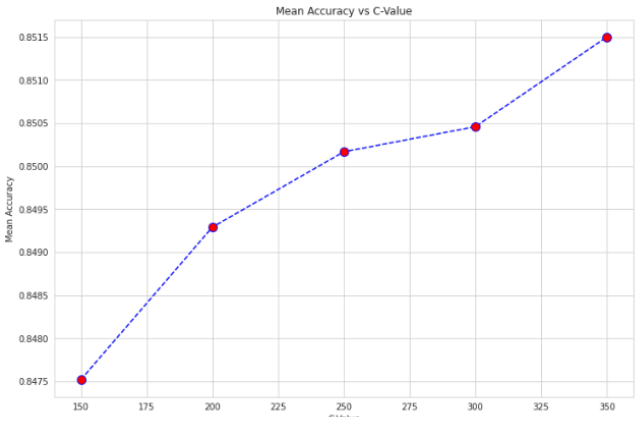
At first we run with less number of n_estimator. So, we run with higher n_estimators again.

```
param_grid2= {'n_estimators': [5, 10, 50, 150, 200],
              'max_depth': [3, 5, 10, None] }
tree_tuner = GridSearchCV(RandomForestClassifier(random_state=42),param_grid2,cv=10, refit=True,verbose=3 )
tree_tuner.fit(X_pca140, np.ravel(y_train))

tree_tuner.best_params_,  tree_tuner.best_estimator_, tree_tuner.best_score_
```

```
param_grid3= {'n_estimators': [150, 200, 250, 300, 350],
              'max_depth': [None] }
tree_tuner1 = GridSearchCV(RandomForestClassifier(random_state=42),param_grid3,cv=10, refit=True,verbose=3 )
tree_tuner1.fit(X_pca140, np.ravel(y_train))

tree_tuner1.best_params_,  tree_tuner1.best_estimator_, tree_tuner1.best_score_
```




```
0.527 (+/-0.015) for {'max_depth': 3, 'n_estimators': 5}
0.557 (+/-0.022) for {'max_depth': 3, 'n_estimators': 10}
0.597 (+/-0.016) for {'max_depth': 3, 'n_estimators': 50}
0.605 (+/-0.013) for {'max_depth': 3, 'n_estimators': 150}
0.610 (+/-0.013) for {'max_depth': 3, 'n_estimators': 200}
0.594 (+/-0.026) for {'max_depth': 5, 'n_estimators': 5}
0.628 (+/-0.014) for {'max_depth': 5, 'n_estimators': 10}
0.671 (+/-0.015) for {'max_depth': 5, 'n_estimators': 50}
0.677 (+/-0.013) for {'max_depth': 5, 'n_estimators': 150}
0.675 (+/-0.015) for {'max_depth': 5, 'n_estimators': 200}
0.698 (+/-0.023) for {'max_depth': 10, 'n_estimators': 5}
0.741 (+/-0.020) for {'max_depth': 10, 'n_estimators': 10}
0.769 (+/-0.021) for {'max_depth': 10, 'n_estimators': 50}
0.777 (+/-0.011) for {'max_depth': 10, 'n_estimators': 150}
0.778 (+/-0.012) for {'max_depth': 10, 'n_estimators': 200}
0.739 (+/-0.014) for {'max_depth': None, 'n_estimators': 5}
0.789 (+/-0.009) for {'max_depth': None, 'n_estimators': 10}
0.838 (+/-0.011) for {'max_depth': None, 'n_estimators': 50}
0.849 (+/-0.011) for {'max_depth': None, 'n_estimators': 150}
0.850 (+/-0.011) for {'max_depth': None, 'n_estimators': 200}
```

Figure: Accuracy Vs n_estimators for Random Forest Classifier.

```
RFC time for training is 464.44170578099875 and time for test is 5.178215595002257

Accuracy: 84.73%

[[2214  141   23    0    1]
 [ 130 1993  260   36    5]
 [  28  193 1855  246   42]
 [   0   30  264 1905  168]
 [   9    8   25  223 2201]]

              precision    recall  f1-score   support

           0       0.93      0.93      0.93      2379
           1       0.84      0.82      0.83      2424
           2       0.76      0.78      0.77      2364
           3       0.79      0.80      0.80      2367
           4       0.91      0.89      0.90      2466

    accuracy                           0.85     12000
   macro avg       0.85      0.85      0.85     12000
weighted avg       0.85      0.85      0.85     12000
```

Gradient Boosting Classifier

A gradient boosted decision tree sometimes yields better accuracy than a random forest. But, it needs more parameter tuning than a random forest. Though it takes a longer time to train, it is faster to test. It also takes up less memory. The two important parameters of the classifier are learning_rate and n_estimators. Learning rate controls correction from the previous tree to the successor and a higher learning rate means stronger correction. N_estimator means the number of trees or weak learners and a higher number generally improves correction. We also tried XGBoost python library to see the performance of gradient boosted decision trees. We found that the prediction accuracy improved for XGBoost by 5%.

```
Gradient Boosting time for training is 1060.809189259 and time for test is 0.387393704002602

Accuracy: 80.12%

[[2153  207   18    0    1]
 [ 194 1861  318   37   14]
 [  41  324 1633  322   44]
 [   0   29  334 1849  155]
 [  16   26   30  275 2119]]

              precision    recall  f1-score   support

           0       0.90      0.91      0.90      2379
           1       0.76      0.77      0.76      2424
           2       0.70      0.69      0.70      2364
           3       0.74      0.78      0.76      2367
           4       0.91      0.86      0.88      2466

    accuracy                           0.80     12000
   macro avg       0.80      0.80      0.80     12000
weighted avg       0.80      0.80      0.80     12000

gb_min

GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=3,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=100,
                           n_iter_no_change=None, presort='auto',
                           random_state=42, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

```
The training time and test time for Gradient Boost are : 1594.186695571, 0.9221335580000414
Accuracy: 85.34%
[[2233  127   18    0    1]
 [ 142 2000  246   33    3]
 [  26  194 1888  225   31]
 [   0   31  277 1910  149]
 [   7   15   24  210 2210]]

              precision    recall  f1-score   support

           0       0.93      0.94      0.93      2379
           1       0.84      0.83      0.83      2424
           2       0.77      0.80      0.78      2364
           3       0.80      0.81      0.81      2367
           4       0.92      0.90      0.91      2466

    accuracy                           0.85     12000
   macro avg       0.85      0.85      0.85     12000
weighted avg       0.85      0.85      0.85     12000

xgb_cl

XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=0, num_parallel_tree=1,
              objective='multi:softprob', random_state=42, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=None, subsample=1,
              tree_method=None, validate_parameters=False, verbosity=None)
```

Figure: Gradient Boosting in scikit-learn and XGBoost.

Voting Classifier

We tried to see how hard voting and soft voting works for the ensemble method. Hard voting aggregates the prediction of each classifier and shows the result for the class that gets the most votes. On the other hand, soft voting counts on the probability of belonging to a class and the highest probability gets the result. We tried the voting classifier with only 4 principal components for doing it faster.

```
VotingClassifier(estimators=[('lr',
                              LogisticRegression(C=1.0, class_weight=None,
                                                 dual=False, fit_intercept=True,
                                                 intercept_scaling=1,
                                                 l1_ratio=None, max_iter=100,
                                                 multi_class='warn',
                                                 n_jobs=None, penalty='l2',
                                                 random_state=None,
                                                 solver='warn', tol=0.0001,
                                                 verbose=0, warm_start=False)),
                             ('rf',
                              RandomForestClassifier(bootstrap=True,
                                                     class_weight=None,
                                                     criterion='gini',...
                                                     oob_score=False,
                                                     random_state=None,
                                                     verbose=0,
                                                     warm_start=False)),
                             ('svc',
                              SVC(C=1.0, cache_size=200, class_weight=None,
                                  coef0=0.0, decision_function_shape='ovr',
                                  degree=3, gamma='auto_deprecated',
                                  kernel='rbf', max_iter=-1, probability=False,
                                  random_state=None, shrinking=True, tol=0.001,
                                  verbose=False))],
                 flatten_transform=True, n_jobs=None, voting='hard',
                 weights=None)
```

```
LogisticRegression 0.4661666666666667
RandomForestClassifier 0.7976666666666666
SVC 0.8125833333333333
VotingClassifier 0.7989166666666667
```

```
VotingClassifier(estimators=[('lr',
                              LogisticRegression(C=1.0, class_weight=None,
                                                 dual=False, fit_intercept=True,
                                                 intercept_scaling=1,
                                                 l1_ratio=None, max_iter=100,
                                                 multi_class='warn',
                                                 n_jobs=None, penalty='l2',
                                                 random_state=None,
                                                 solver='warn', tol=0.0001,
                                                 verbose=0, warm_start=False)),
                             ('rf',
                              RandomForestClassifier(bootstrap=True,
                                                     class_weight=None,
                                                     criterion='gini',...
                                                     oob_score=False,
                                                     random_state=None,
                                                     verbose=0,
                                                     warm_start=False)),
                             ('svc',
                              SVC(C=1.0, cache_size=200, class_weight=None,
                                  coef0=0.0, decision_function_shape='ovr',
                                  degree=3, gamma='auto_deprecated',
                                  kernel='rbf', max_iter=-1, probability=True,
                                  random_state=None, shrinking=True, tol=0.001,
                                  verbose=False))],
                 flatten_transform=True, n_jobs=None, voting='soft',
                 weights=None)
```

```
LogisticRegression 0.4661666666666667
RandomForestClassifier 0.7949166666666667
SVC 0.8125833333333333
VotingClassifier 0.8128333333333333
```

Figure: Hard Voting and Soft Voting classifiers.

AdaBoost

AdaBoost means Adapting Boosting. It is also an ensemble that combines multiple classifiers to predict better accuracy by increasing weights for the subsequent classifier. It is a sequential learning technique that improves upon the predecessor's misclassification. Any classifier can be used to AdaBoost. We have tried AdaBoost with decision tree classifier and Random Forest Classifier.

One of the drawbacks of the AdaBoost is that it sequential, so it cannot be parallelized like bagging and it takes a long time.

Important parameters for AdaBoost are base_estimator, n_estimators, and learning_rate.

Base_estimator: random forest classifier and decision tree classifier
N_estimaor : 50, it controls the number of weak learners
Learning_rate: 1, it controls the contribution of the weak learners in the final combination.

1.2 Implementation of your Design Choices (5 points)

(Show some of the important code blocks to implement your model. We will also consult your full code on LEARN, so this is your chance to guide us to understand your code and how you achieved your result)

Codes have been shown in the snapshots as the classifier used are discussed.

Random Forest Classifier

In [ ]:
```
param_grid_rfc= {'n_estimators': [5, 10, 50, 150, 200, 300, 350, 650, 850],'max_depth': [3, 5, 10, None] }
```

In [ ]:
```
tree_tuner = GridSearchCV(RandomForestClassifier(random_state=42),param_grid_rfc,cv=10, refit=True,verbose=3 )
tree_tuner.fit(X_pca74, np.ravel(y_train))

tree_tuner.best_params_,  tree_tuner.best_estimator_, tree_tuner.best_score_
```

Plotting the accuracy against n_estimators

In [ ]:
```
sns.set_style('whitegrid')
plt.figure(figsize=(12,8))
plt.plot(tree_tuner.param_grid_rfc['n_estimators'], tree_tuner.cv_results_['mean_test_score'],color='blue',
         linestyle='dashed', marker='o', markerfacecolor='red', markersize=10 )
plt.title('Mean Accuracy vs Maximum Depth')
plt.xlabel('Maximum Depth')
plt.ylabel('Mean Accuracy')
```

Printing accuracy agains the hyperparameters

In [ ]:
```
print("Gridsearch scores on C value:")
print()
means = tree_tuner.cv_results_['mean_test_score']
stds = tree_tuner.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, tree_tuner.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std*2 , params))
print()
```

Random Forest Classifier Experiment: Best parameters, Metrics, Time Analysis

In [60]:
```
import timeit

rfc_min1 = RandomForestClassifier(n_estimators= 650, random_state= 42)

# Training time
start = timeit.default_timer()
rfc_min1.fit(X_train, np.ravel(y_train))
stop = timeit.default_timer()
time_rfc_min = stop - start

# Test Time
start_testrfc = timeit.default_timer()
pred_rfc_min1 = rfc_min1.predict(X_val)
stop_testrfc = timeit.default_timer()
time_testrfc = stop_testrfc - start_testrfc

print ("RFC time for training is {} and time for test is {}".format(time_rfc_min, time_testrfc))

acc_rfc = accuracy_score(y_val, pred_rfc_min1)

print('\n')
print("Accuracy: %.2f%%" % (acc_rfc * 100.0))
print('\n')
print(confusion_matrix(y_val,pred_rfc_min1))
print('\n')
print(classification_report(y_val,pred_rfc_min1))
```

```
RFC time for training is 480.90744357400035 and time for test is 4.452418802000466


Accuracy: 84.82%
```

KNN Classifier

[79]:
```
grid_params = {'n_neighbors' : [3,5,11,19, 25, 29, 35, 45, 51],
               'weights': ['uniform', 'distance'],
               'metric': ['euclidean', 'manhattan']}
```

[80]:
```
X_pca140.shape
```

t[80]:
```
(48000, 140)
```

[81]:
```
model_tuner = GridSearchCV(KNeighborsClassifier(),grid_params, cv=3, refit=True,verbose=3)
model_tuner.fit(X_pca140, np.ravel(y_train))
model_tuner.best_params_,  model_tuner.best_estimator_, model_tuner.best_score_
```

```
[CV]  metric=euclidean, n_neighbors=35, weights=uniform, score=0.845, total=  45.6s
[CV] metric=euclidean, n_neighbors=35, weights=distance .............
[CV]  metric=euclidean, n_neighbors=35, weights=distance, score=0.847, total=  50.4s
[CV] metric=euclidean, n_neighbors=35, weights=distance .............
[CV]  metric=euclidean, n_neighbors=35, weights=distance, score=0.840, total=  51.3s
[CV] metric=euclidean, n_neighbors=35, weights=distance .............
[CV]  metric=euclidean, n_neighbors=35, weights=distance, score=0.848, total=  51.3s
[CV] metric=euclidean, n_neighbors=45, weights=uniform ..............
[CV]  metric=euclidean, n_neighbors=45, weights=uniform, score=0.841, total=  55.9s
[CV] metric=euclidean, n_neighbors=45, weights=uniform ..............
[CV]  metric=euclidean, n_neighbors=45, weights=uniform, score=0.835, total= 1.0min
[CV] metric=euclidean, n_neighbors=45, weights=uniform ..............
[CV]  metric=euclidean, n_neighbors=45, weights=uniform, score=0.842, total= 1.0min
[CV] metric=euclidean, n_neighbors=45, weights=distance .............
[CV]  metric=euclidean, n_neighbors=45, weights=distance, score=0.844, total=  51.4s
[CV] metric=euclidean, n_neighbors=45, weights=distance .............
[CV]  metric=euclidean, n_neighbors=45, weights=distance, score=0.838, total=  51.8s
```

97]:
```
model_tuner.param_grid['n_neighbors'], model_tuner.cv_results_['mean_test_score']
```

97]:
```
([3, 5, 11, 19, 25, 29, 35, 45, 51],
 array([0.83339583, 0.83545833, 0.84127083, 0.84410417, 0.8466875 ,
        0.84958333, 0.84575   , 0.84833333, 0.84520833, 0.84727083,
        0.84304167, 0.84545833, 0.84195833, 0.8449375 , 0.839375  ,
        0.8423125 , 0.83808333, 0.8409375 , 0.79604167, 0.802125  ,
        0.808875  , 0.81258333, 0.81695833, 0.82104167, 0.81825   ,
        0.82108333, 0.81625   , 0.81947917, 0.815     , 0.81827083,
        0.81120833, 0.815375  , 0.80847917, 0.81227083, 0.80525   ,
        0.8095625 ]))
```

Printing the result against K-value

83]:
```
print("Gridsearch scores on K value:")
print()
means = model_tuner.cv_results_['mean_test_score']
stds = model_tuner.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, model_tuner.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std*2 , params))
print()
```

```
Gridsearch scores on K value:

0.833 (+/-0.002) for {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'uniform'}
0.835 (+/-0.002) for {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'distance'}
0.841 (+/-0.006) for {'metric': 'euclidean', 'n_neighbors': 5, 'weights': 'uniform'}
0.844 (+/-0.005) for {'metric': 'euclidean', 'n_neighbors': 5, 'weights': 'distance'}
0.847 (+/-0.006) for {'metric': 'euclidean', 'n_neighbors': 11, 'weights': 'uniform'}
0.850 (+/-0.005) for {'metric': 'euclidean', 'n_neighbors': 11, 'weights': 'distance'}
0.846 (+/-0.006) for {'metric': 'euclidean', 'n_neighbors': 19, 'weights': 'uniform'}
0.848 (+/-0.005) for {'metric': 'euclidean', 'n_neighbors': 19, 'weights': 'distance'}
0.845 (+/-0.009) for {'metric': 'euclidean', 'n_neighbors': 25, 'weights': 'uniform'}
0.847 (+/-0.009) for {'metric': 'euclidean', 'n_neighbors': 25, 'weights': 'distance'}
0.843 (+/-0.007) for {'metric': 'euclidean', 'n_neighbors': 29, 'weights': 'uniform'}
0.845 (+/-0.007) for {'metric': 'euclidean', 'n_neighbors': 29, 'weights': 'distance'}
0.842 (+/-0.007) for {'metric': 'euclidean', 'n_neighbors': 35, 'weights': 'uniform'}
0.845 (+/-0.007) for {'metric': 'euclidean', 'n_neighbors': 35, 'weights': 'distance'}
0.839 (+/-0.007) for {'metric': 'euclidean', 'n_neighbors': 45, 'weights': 'uniform'}
0.842 (+/-0.007) for {'metric': 'euclidean', 'n_neighbors': 45, 'weights': 'distance'}
0.838 (+/-0.007) for {'metric': 'euclidean', 'n_neighbors': 51, 'weights': 'uniform'}
0.841 (+/-0.007) for {'metric': 'euclidean', 'n_neighbors': 51, 'weights': 'distance'}
0.796 (+/-0.004) for {'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'uniform'}
0.802 (+/-0.006) for {'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'distance'}
```

KNN Experiment: Best parameters, Metrics, Time Analysis

```
]: import timeit

knn_min1 = KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='euclidean',
                                metric_params=None, n_jobs=None, n_neighbors=11, p=2, weights='distance')

# Training time
start = timeit.default_timer()
knn_min1.fit(X_train_min, np.ravel(y_train))
stop = timeit.default_timer()
time_knn_min = stop - start

# Test Time
start_testknn = timeit.default_timer()
pred_knn_min1 = knn_min1.predict(X_val_min)
stop_testknn = timeit.default_timer()
time_testknn = stop_testknn - start_testknn

print ("KNN time for training is {} and time for test is {}".format(time_knn_min, time_testknn))

acc_svm = accuracy_score(y_val, pred_knn_min1)

print('\n')
print("Accuracy: %.2f%%" % (acc_svm * 100.0))
print('\n')
print(confusion_matrix(y_val,pred_knn_min1))
print('\n')
print(classification_report(y_val,pred_knn_min1))
```

```
KNN time for training is 36.583538853999926 and time for test is 811.5477682929995


Accuracy: 84.88%
```

SVM Classifier

Experiment with Linear Kernel

```
[ ]: import timeit

svm = SVC(kernel='linear', random_state= 42)

start = timeit.default_timer()
svm.fit(X_pca74, np.ravel(y_train))
stop = timeit.default_timer()
time_train_svc = start - stop


start_t = timeit.default_timer()
pred_svm = svm.predict(X_val_pca74)
stop_t = timeit.default_timer()
time_test_svc = stop_t - start_t

print ("SVM time for training is {} and time for test is {}".format(time_train_svc, time_test_svc))
print(confusion_matrix(y_val,pred_svm))
print('\n')
print(classification_report(y_val,pred_svm))
```

Let's do a gridsearchCV for important hyperparameters for a SVM classifier

```
99]: param_grid_svc = {'C': [0.1, 0.5, 1, 5, 15, 35, 50], 'kernel': ['rbf'], 'gamma': [1,0.1,0.01,0.001]}
```

```
[*]: grid_svm = GridSearchCV(SVC(random_state=42),param_grid_svc,cv=10, refit=True,verbose=3)
grid_svm.fit (X_pca74, np.ravel(y_train))
grid_svm.best_params_,  grid_svm.best_estimator_, grid_svm.best_score_
```

```
[CV] C=5, gamma=0.001, kernel=rbf ....................................
[CV] ........ C=5, gamma=0.001, kernel=rbf, score=0.796, total= 2.0min
[CV] C=5, gamma=0.001, kernel=rbf ....................................
[CV] ........ C=5, gamma=0.001, kernel=rbf, score=0.796, total= 2.0min
[CV] C=5, gamma=0.001, kernel=rbf ....................................
[CV]          C=5, gamma=0.001, kernel=rbf, score=0.811, total= 2.0min
```

SVM Experiment: Best parameters, Metrics, Time Analysis

```
]: from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix,classification_report

import timeit

svm_min1 = SVC(kernel='rbf', C=50, random_state= 42)

# Training time
start = timeit.default_timer()
svm_min1.fit(X_train_min, np.ravel(y_train))
stop = timeit.default_timer()
time_svm_min = stop - start

# Test Time
start_testsvm = timeit.default_timer()
pred_svm_min1 = svm_min1.predict(X_val_min)
stop_testsvm = timeit.default_timer()
time_testsvm = stop_testsvm - start_testsvm

print ("SVM time for training is {} and time for test is {}".format(time_svm_min, time_testsvm))

acc_svm = accuracy_score(y_val, pred_svm_min1)

print("Accuracy: %.2f%%" % (acc_svm * 100.0))

print(confusion_matrix(y_val,pred_svm_min1))
print('\n')
print(classification_report(y_val,pred_svm_min1))
```

```
/Users/mokul791/anaconda3/lib/python3.7/site-packages/sklearn/svm/base.py:193: FutureWarning: The defaul
gamma will change from 'auto' to 'scale' in version 0.22 to account better for unscaled features. Set ga
tly to 'auto' or 'scale' to avoid this warning.
  "avoid this warning.", FutureWarning)
```

```
SVM time for training is 856.8173299270002 and time for test is 244.31789683500028
Accuracy: 85.27%
```

Gradient Boosting Classifier with XGBoost

Gradient Boosting Classifier Experiment with XGBoost : Best parameters, Metrics, Time Analysis

```
import timeit
xgb_cl = xgb.XGBClassifier(random_state= 42)

#Training Time
start = timeit.default_timer()
xgb_cl.fit(X_train, np.ravel(y_train))
stop = timeit.default_timer()
time_gbtrain = stop - start

# Testing Time

start_gbtest = timeit.default_timer()
y_pred_gb = xgb_cl.predict(X_val)
stop_gbtest = timeit.default_timer()
test_gbtest = stop_gbtest - start_gbtest


print ("The training time and test time for Gradient Boost are : {}, {} ".format(time_gbtrain, test_gbtest))

# Result
accuracy = accuracy_score(y_val, y_pred_gb)
print("Accuracy: %.2f%%" % (accuracy * 100.0))

print(confusion_matrix(y_val,y_pred_gb))
print('\n')
print(classification_report(y_val,y_pred_gb))
```

```
The training time and test time for Gradient Boost are : 1594.186695571, 0.9221335580000414
Accuracy: 85.34%
```

ROC for Random Forest

```
47]: class_rfc = OneVsRestClassifier(RandomForestClassifier(n_estimators=650, random_state= 42))
```

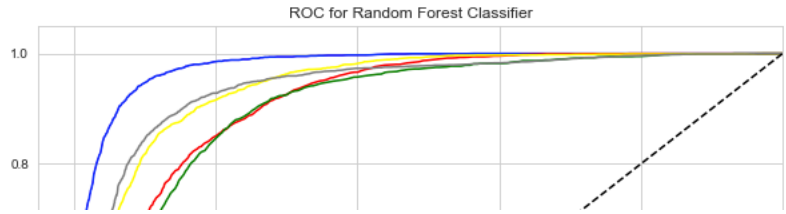
```
48]: y_score_rfc = class_rfc.fit(X_pca140_roc, y_train_roc).predict_proba(X_val_pca140_roc)
```

```
50]: n_classes = y_bin.shape[1]
     fpr = dict()
     tpr = dict()
     roc_auc = dict()
     for i in range(n_classes):
         fpr[i], tpr[i], _ = roc_curve(y_val_roc[:, i], y_score_rfc[:, i])
         roc_auc[i] = auc(fpr[i], tpr[i])
     colors = ('blue', 'red', 'green', 'yellow', 'gray')

     plt.figure(figsize=(10, 8))
     plt.savefig('DTC_ROC and AUC.pdf')

     for i, color in zip(range(n_classes), colors):
         plt.plot(fpr[i], tpr[i], color=color,
                  label='ROC curve of class {0} (area = {1:0.2f})'
                  ''.format(i, roc_auc[i]))


     plt.plot([0, 1], [0, 1], 'k--')
     plt.xlim([-0.05, 1.0])
     plt.ylim([0.0, 1.05])
     plt.xlabel('False Positive Rate')
     plt.ylabel('True Positive Rate')
     plt.title('ROC for Random Forest Classifier')
     plt.legend(loc="lower right")
     plt.show()
```



Time Analysis of different classifier

Time for different classifiers were save in a local excel file for easy comparision

]:
```
time_df = pd.read_excel('time.xlsx')
```

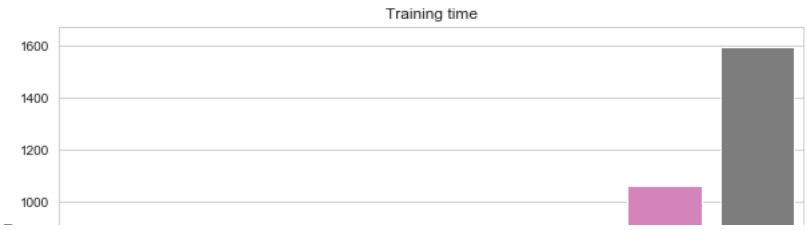
]:
```
time_df
```

]:

	Classifier	Training	Test
0	KNN	36.58	811.55
1	SVM	856.82	244.32
2	DTC	18.69	0.13
3	DTC+Ada	897.04	1.54
4	RFC	480.90	4.45
5	RFC+Ada	452.35	3.67
6	GB+SL	1060.80	0.38
7	GB+Xgb	1594.18	0.92

]:
```
plt.figure(figsize=(10,6))
sns.barplot( x = 'Classifier',y = 'Training', data=time_df)
plt.title('Training time')
```

]:
```
Text(0.5, 1.0, 'Training time')
```



1.3 Kaggle Competition Score (5 points)

(Report the highest score your submissions received on Kaggle. If you have any explanations for varying performance by different teams explain it here)

Our highest score for non-neural network classifiers is 84.5%. Initially, we received a low accuracy of 70% and 74% with PCA components of 140. But, after we evaluated the results of various classifiers, we trained the model with all features. Thus, we have received an improvement in prediction accuracy. Our analysis is that as our data is images, generally neural network-based classifier such as CNN performs better than classical machine learning algorithms. Most of the classical machine learning does not extract the features the way deep learning algorithms do. CNN, for example, can build an internal representation of the images. The other reason we think is that extraction is layered and weights are refined at every layer.

1.4 Result Analysis: Runtime Performance for training and testing

The summary of the runtime performance of training and testing for all the classifiers that have tried is given in the following table. The result of running the models are given at figures. The explanations for runtime performance are given under the head of classifiers.

Classifier	Training in Big O	Experimental	Testing in Big O	Experimental
KNN	$O(1)$	36.58	$O(np)$	811.55
SVM	Between $O(n^2 \times p)$ and $O(n^3 \times p)$	856.82	$O(n)$	244. 32
DTC	$O(p \times n(\log_2 n)$	18.69	$O(1)$	0.13
DTC with AdaBoost		897.04	$O(1)$	1.54
Random Forest Classifier (RFC)	$O(n_{tree} \times n(\log_2 n))$	480.90	$O(1)$	4.45
RFC with AdaBoost		452.35	$O(1)$	3.67
Gradient Boosting with Scikit-learn	$O(n_{tree} \times n(\log_2 n))$	1060.80	$O(1)$	0.38
Gradient Boosting with Xgboost		1594.18	$O(1)$	.92

Table: Summary of runtime performance of training and testing of classifiers.(n is number of samples, p= number of features, n_tree = number of tree, depth= height of tree)

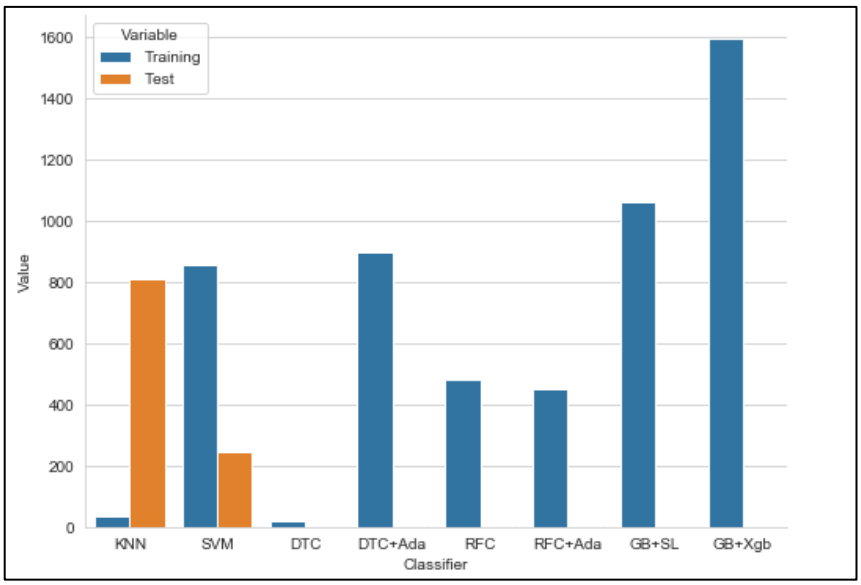


Figure: Summary of runtime performance of training and testing of classifiers.

KNN time for training is 36.583538853999926 and time for test is 811.5477682929995

Accuracy: 84.88%

```
[[2280   92    6    1    0]
 [ 202 2005  162   50    5]
 [ 102  251 1763  205   43]
 [   2   41  241 1935  148]
 [  18   10   32  203 2203]]
```

	precision	recall	f1-score	support
0	0.88	0.96	0.92	2379
1	0.84	0.83	0.83	2424
2	0.80	0.75	0.77	2364
3	0.81	0.82	0.81	2367
4	0.92	0.89	0.91	2466
accuracy			0.85	12000
macro avg	0.85	0.85	0.85	12000
weighted avg	0.85	0.85	0.85	12000

knn_min1

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='euclidean',
                     metric_params=None, n_jobs=None, n_neighbors=11, p=2,
                     weights='distance')
```

SVM time for training is 856.8173299270002 and time for test is 244.31789683500028

Accuracy: 85.27%

```
[[2235  118   26    0    0]
 [ 192 1987  214   25    6]
 [  39  238 1865  199   23]
 [   0   34  261 1931  141]
 [  12   16   35  189 2214]]
```

	precision	recall	f1-score	support
0	0.90	0.94	0.92	2379
1	0.83	0.82	0.82	2424
2	0.78	0.79	0.78	2364
3	0.82	0.82	0.82	2367
4	0.93	0.90	0.91	2466
accuracy			0.85	12000
macro avg	0.85	0.85	0.85	12000
weighted avg	0.85	0.85	0.85	12000

svm_min1

```
SVC(C=50, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    kernel='rbf', max_iter=-1, probability=False, random_state=42,
    shrinking=True, tol=0.001, verbose=False)
```

DTC time for training is 18.69547382000019 and time for test is 0.1251100299996324

Accuracy: 71.90%

```
[[2025  293   53    2    6]
 [ 297 1655  403   49   20]
 [  76  429 1409  390   60]
 [   6   75  384 1638  264]
 [  16   49  105  395 1901]]
```

	precision	recall	f1-score	support
0	0.84	0.85	0.84	2379
1	0.66	0.68	0.67	2424
2	0.60	0.60	0.60	2364
3	0.66	0.69	0.68	2367
4	0.84	0.77	0.81	2466
accuracy			0.72	12000
macro avg	0.72	0.72	0.72	12000
weighted avg	0.72	0.72	0.72	12000

dtc_min1

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=12,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=42, splitter='best')
```

DTC with AdaBoost time for training is 897.0444833190013 and time for test is 1.5386785619994043

Accuracy: 80.39%

```
[[2109  236   33    0    1]
 [ 119 1935  335   32    3]
 [  35  257 1808  235   29]
 [   0   37  385 1804  141]
 [  12   37   45  381 1991]]
```

	precision	recall	f1-score	support
0	0.93	0.89	0.91	2379
1	0.77	0.80	0.79	2424
2	0.69	0.76	0.73	2364
3	0.74	0.76	0.75	2367
4	0.92	0.81	0.86	2466
accuracy			0.80	12000
macro avg	0.81	0.80	0.81	12000
weighted avg	0.81	0.80	0.81	12000

abc_dtc

```
AdaBoostClassifier(algorithm='SAMME.R',
                   base_estimator=DecisionTreeClassifier(class_weight=None,
                                                         criterion='gini',
                                                         max_depth=12,
                                                         max_features=None,
                                                         max_leaf_nodes=None,
                                                         min_impurity_decrease=0.0,
                                                         min_impurity_split=None,
                                                         min_samples_leaf=1,
                                                         min_samples_split=2,
                                                         min_weight_fraction_leaf=0.0,
                                                         presort=False,
                                                         random_state=42,
                                                         splitter='best'),
                   learning_rate=1, n_estimators=50, random_state=None)
```

RFC time for training is 452.35469498100065 and time for test is 3.673787560001074

Accuracy: 84.58%

```
[[2214  142   22    0    1]
 [ 129 1990  264   36    5]
 [  28  196 1844  252   44]
 [   0   32  257 1905  173]
 [   9   10   25  226 2196]]
```

	precision	recall	f1-score	support
0	0.93	0.93	0.93	2379
1	0.84	0.82	0.83	2424
2	0.76	0.78	0.77	2364
3	0.79	0.80	0.80	2367
4	0.91	0.89	0.90	2466
accuracy			0.85	12000
macro avg	0.85	0.85	0.85	12000
weighted avg	0.85	0.85	0.85	12000

RFC time for training is 480.90744357400035 and time for test is 4.452418802000466

Accuracy: 84.82%

```
[[2215  142   21    0    1]
 [ 130 1995  258   36    5]
 [  24  195 1860  244   41]
 [   0   30  256 1911  170]
 [   9   10   24  226 2197]]
```

	precision	recall	f1-score	support
0	0.93	0.93	0.93	2379
1	0.84	0.82	0.83	2424
2	0.77	0.79	0.78	2364
3	0.79	0.81	0.80	2367
4	0.91	0.89	0.90	2466
accuracy			0.85	12000
macro avg	0.85	0.85	0.85	12000
weighted avg	0.85	0.85	0.85	12000

rfc_min1

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=650,
                       n_jobs=None, oob_score=False, random_state=42, verbose=0,
                       warm_start=False)
```

Gradient Boosting time for training is 1060.809189259 and time for test is 0.3873937040002602

Accuracy: 80.12%

```
[[2153  207   18    0    1]
 [ 194 1861  318   37   14]
 [  41  324 1633  322   44]
 [   0   29  334 1849  155]
 [  16   26   30  275 2119]]
```

	precision	recall	f1-score	support
0	0.90	0.91	0.90	2379
1	0.76	0.77	0.76	2424
2	0.70	0.69	0.70	2364
3	0.74	0.78	0.76	2367
4	0.91	0.86	0.88	2466
accuracy			0.80	12000
macro avg	0.80	0.80	0.80	12000
weighted avg	0.80	0.80	0.80	12000

gb_min

```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=3,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=100,
                           n_iter_no_change=None, presort='auto',
                           random_state=42, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

The training time and test time for Gradient Boost are : 1594.186695571, 0.9221335580000414

Accuracy: 85.34%

```
[[2233  127   18    0    1]
 [ 142 2000  246   33    3]
 [  26  194 1888  225   31]
 [   0   31  277 1910  149]
 [   7   15   24  210 2210]]
```

	precision	recall	f1-score	support
0	0.93	0.94	0.93	2379
1	0.84	0.83	0.83	2424
2	0.77	0.80	0.78	2364
3	0.80	0.81	0.81	2367
4	0.92	0.90	0.91	2466
accuracy			0.85	12000
macro avg	0.85	0.85	0.85	12000
weighted avg	0.85	0.85	0.85	12000

xgb_cl

```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=0, num_parallel_tree=1,
              objective='multi:softprob', random_state=42, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=None, subsample=1,
              tree_method=None, validate_parameters=False, verbosity=None)
```

Figure: Runtime performance of training and testing of KNN, SVM, Decision Tree Classifier with and without AdaBoost, Random Forest Classifier with and without AdaBoost, Gradient Boosting with scikit-learn and XGBoost

- **KNN** As we know KNN classifies the test data based on the neighbors' class calculating a distance function, thus it does not in a real sense of learning something. Because it does not correct itself by any feedback that we use in neural networks. For the training, a KNN model just stores the data points in memory without doing any calculation. Therefore, running time for training the model for KNN is less, as it just needs to store the value which is done in space complexity of $O(1)$. But, when it comes to predicting the test data, it does the calculation. The running time is $O(np)$, where n = number of sample and p= number of features i.e. it has to compare every single data point in the training data. Thus, testing time is much larger and in our case, as our dataset consists of 60000 samples, it took a long time. Finally, it shows that time for the algorithm to run is not upfront spent on training, rather it is spent on testing, i.e. predicting.

- **SVM** We used the scikit-learn SVC classifier that supports kernel tricks. The runtime performance of SVM is large. For training, the time complexity is large, between $O(m^2 \times n)$ and $O(m^3 \times n)$. Thus, as our training samples were large, 60000, the time for training was huge. However, as we used reduced features through PCA, from 784 to 140, we could reduce the value of n and got a better time for training with reduced features. Again, unlike our previous experiment with digit images which was sparse, our data were dense vectors because most of the images were full on the 28 X 28 scale.

- **Decision Tree Classifier** We used scikit-learn, so it used the CART (Classification and Regression Tree) algorithm that produces only binary trees. So, it was pure yes and no. It is based on the greedy algorithm that makes a choice in each split taking the best solution, but it is not an optimal solution. As the optimal solution is an NP-complete problem, that has a runtime of O(exp(n)), this approach is quite faster, the runtime of $O(\log_2 n) - j$ just as a binary tree algorithm. In our model, we kept the max-feature at NONE, meaning it compared all features. Therefore, our runtime complexity was $O(m \times n(\log_2 n)$. We also set the presort to default value i.e. False, so our training ran on $O(m \times n(\log_2 n)$ time complexity.

- **Random Forest Classifier**. Random Forest classifier is computationally large. To reduce the computational energy of the computer, we set the n_jobs to default value NONE which is 1. It took more memory and time to train and predict than linear models.

1.4 Result Analysis: Comparison of the different algorithms and parameters you tried.

Classifier / Algorithms	Important Parameters	Best Parameter Found	Accuracy
KNN	• n_neighbors = [3,5,11,19, 25, 29, 35, 45, 51] • weights = ['uniform', 'distance'] • metric = ['euclidean', 'manhattan']	n_neighbors = 11 weights = 'distance' metric = 'euclidean'	84.4%
SVM	• kernel = ['linear', 'rbf'] • C = [0.1, 0.5, 1, 5, 15, 35, 50] • gamma = [1,0.1,0.01,0.001]	kernel: 'rbf' C = 15 Gamma = 0.01 (scikit-learn 'Scale' = 1 / (n_features * X.var()). For our case: 1/ (784 * .0.1244)	84%
Decision Tree Classifier	• criterion= ['gini', 'entropy'] • max_depth = [4, 6, 8, 12, None]	criterion= 'gini' max_depth = 12	72%
Decision Tree with AdaBoost	• Decision Tree same parameters • AdaBoost = Default		80%
Random Forest Classifier	• n_estimators = [5, 10, 50, 150, 200, 300, 350, 650, 850] • max_depth= [3, 5, 10, None]	n_estimators = 650 max_depth = None	85%
Random Forest with AdaBoost	• RFC same parameters • AdaBoost = Default		85%
Gradient Boosting	• learning_rate = 0.1 • n_estimator = 100	learning_rate = 0.1 n_estimator = 100	80%
Gradient Boosting with XGBoost	• learning_rate = 0.3 • n_estimator = 100		85%

Explanations and comparison of the tested algorithms are given as an answer to the previous questions. Here only parameters are repeated.

- **KNN** Out of the number of non-neural network algorithms that we used, KNN is one of the simplest. It predicts the class of a data point by calculating the distance from the training data points and assign the classes by voting by neighbors' classes. It does not learn, meaning that it does not have any feedback to correct its prediction. Three parameters are important determinant of the better result of KNN. They are the value of k (number of neighbors for calculating distance), weights (for voting), and distance metric.

- n_neighbors: The value of k is important. If the value is too small, it gets influenced by noise and outlier data points. It has a low bias but a very high variance. Again, if k is larger, it increases the bias but lowers the variance. Overall, the k-value parameter defines the flexibility of the model where lower value provides an overly flexible model and a very higher value makes the model less flexible. That is why we needed to train the model with a range of k-values, [3,5,11,19, 25, 29, 35, 45, 51], to get the optimum k-value for a specific dataset.

- Metric: While the most popular distance metric is Euclidean or L2 – a distance that just calculates the distance following the Pythagorean theorem, we also tested 'Manhattan' distance or L1- distance. It made more sense as the straight line distance measure provided better result than the block-based distance. The straight line distance measure provided better result than the block-based distance.

- weights: For the voting weights, we tried both ‘uniform’ and ‘distance’. In uniform weight, regardless of the distance between the training data point and the test data point, the voting weight is the same. Thus, it does not provide any voting weight. But, in the case of ‘distance’ weight, the voting weight is affected by the distance of the training data point.

- **SVM** The important hyperparameters of SVM are the regularization parameter (C parameter), kernel function, and gamma parameter.

- Regulation Parameter (C parameter): C parameter deals with the misclassification of the training data points. It dictates the soft margin classification. Larger C value provides the smaller-margin hyperplane, but accept less margin violation, and lower C value provides the larger-margin for hyperplane but accepts more margin violation. Thus, higher C value allows each data point to have greater influence in the model for classification boundary. For our model, we used seven C value [0.1, 0.5, 1, 5, 15, 35, 50].

- Kernel Parameter: We tried with two kernels, ‘linear’ and ‘rbf’. The linear model is generally better when the number of features is large compared to the training samples. But, as our training samples are large, the linear kernel took a significant amount of time. Since our dataset is nonlinear, it made sense to use ‘rbf’ which is defined by the Gaussian Radial Basis Function. One of the disadvantages of the rbf kernel is that it transformed our 60000 samples and 784 features into a training set of 60000 samples and 60000 features.

- Gamma: For SVM with ‘rbf’, gamma is an important hyperparameter. The hyperparameter gamma influences the shape/ width of the gaussian curve. With high gamma value, the curve becomes narrower and thus the decision boundary adjusts with each class data point. With smaller gamma value, the shape of the curve becomes wider and the decision boundary becomes smoother.

- **Decision Tree Classifier** Since post-pruning is not available in scikit-learn, so we tried to pre-pruning by limiting the maximum depth. The important regularization parameter for the decision tree classifier is the max_depth. This hyperparameter limits the CART algorithm to stop splitting the training set. So, it acted as a stopping criterion for the running of our model. This is also essential for limiting the overfitting of the training data. We uses a combination of max_depth = [4, 6, 8, 12, None].

  Another important parameter of the decision tree classifier is the criterion for the information impurity. In scikit-learn, we had two choices for information integrity or for reducing the impurity, gini and entropy. For example, a node is pure if its gini is 0. However, they both often result in a similar tree. Since our classes were well distributed, gini was a better choice. It runs faster than entropy. However, entropy would be a good choice if our data classes were not balanced, as it leads to a more balanced tree. We run on both criteria.

- **Random Forest Classifier**

- One of the important parameters of the random forest classifier is the max_feature. For our case, we used auto, i.e. n_features. So, in each split, it looked at all the features while maintaining the bootstrap property. Lower max_feature builds more diverse trees and higher max_features build more similar trees.

- 'n_estimators': [5, 10, 50, 150, 200, 650, 850],. The larger is always better as averaging more trees provide better probability and reduce the overfitting. But, a higher number of trees require more memory that we experienced while running, taking almost all the storage of the computer. It also took more time.

- 'max_depth': [3, 5, 10, None] }

- Bootstrap = true, so the BootStrap aggregating property maintained.

- Criterion = gini for decision tree information purity.

- Random State: Since Random Forest is highly random, it is essential to fix the random state to reproduce the result of the model. We set it to 42.

- **Gradient Boosting Classifier** The two important parameters of the classifier are learning_rate and n_estimators. Learning rate controls correction from the previous tree to the successor and a higher learning rate means stronger correction. N_estimator means the number of trees or weak learners and a higher number generally improves correction. We also tried XGBoost python library to see the performance of gradient boosted decision trees. We found that the prediction accuracy improved for XGBoost by 5%.

- **AdaBoost** Important parameters for AdaBoost are base_estimator, n_estimators, and learning_rate.

- Base_estimator: random forest classifier and decision tree classifier.
- N_estimaor : 50, it controls the number of weak learners.
- Learning_rate: 1, it controls the contribution of the weak learners in the final combination.

1.4 Result Analysis : Explanation of your model (algorithms, design choices, numbers of parameters).

We found the best accuracy with Random Forest Classifier. However, we also noticed an almost equally good result for SVM. Our model is explained in the following paragraphs.

Random Forest classifier is an ensemble method and is a multi-class classifier. It performs well when the individual classifiers are independent of each other, thus not making the same error and improving the accuracy of classification. It uses the same algorithm as a decision tree classifier, but, it trains the models with different subsets of the training samples. This is often done with a replacement which is known as bootstrap aggregating. So, a random forest classifier combines the bagging classifier and decision tree classifier. It uses soft-voting for classification, i.e. it provides more weights on the best probability of the classifiers.

The problem of overfitting in Decision tree classifiers can be addressed by the Random Forest classifier by building multiple decision trees and vote them to classify. Thus, it avoids the overfitting by the decision tree by averaging the decision tree classifiers' results. It does by building random decision trees. Another advantage of the Random forest is that it captures a broader picture of the data than any other single tree. It does not require any scaling of the data.

However, the Random Forest classifier models are difficult to interpret, that is to say, that it is a sort of black-box model like neural network models. Because it builds a large number of trees, it is difficult to visualize and explain.

We tested with the following parameters to find out the best hyperparameters for our mode:

- One of the important parameters of the random forest classifier is the max_feature. For our case, we used auto, i.e. n_features. So, in each split, it looked at all the features while maintaining the bootstrap property. Lower max_feature builds more diverse trees and higher max_features build more similar trees.

- 'n_estimators': [5, 10, 50, 150, 200, 650, 850],. The larger is always better as averaging more trees provide better probability and reduce the overfitting. But, a higher number of trees require more memory that we experienced while running, taking almost all the storage of the computer. It also took more time.

- 'max_depth': [3, 5, 10, None] }

- Bootstrap = true, so the BootStrap aggregating property maintained.

- Criterion = gini for decision tree information purity.
- Random State: Since Random Forest is highly random, it is essential to fix the random state to reproduce the result of the model. We set it to 42.

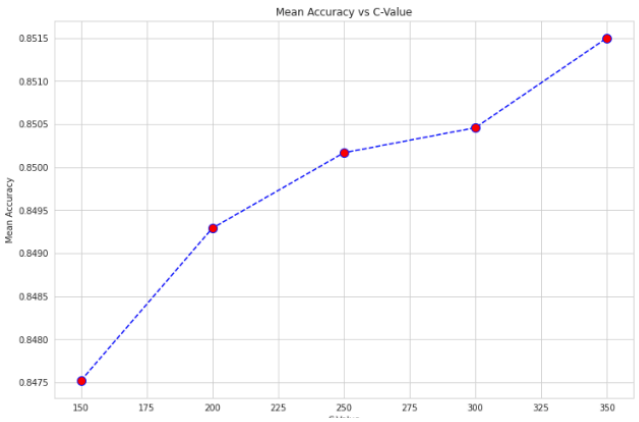
At first we run with less number of n_estimator. So, we run with higher n_estimators again.

```
param_grid2= {'n_estimators': [5, 10, 50, 150, 200],
              'max_depth': [3, 5, 10, None] }
tree_tuner = GridSearchCV(RandomForestClassifier(random_state=42),param_grid2,cv=10, refit=True,verbose=3 )
tree_tuner.fit(X_pca140, np.ravel(y_train))

tree_tuner.best_params_,  tree_tuner.best_estimator_, tree_tuner.best_score_
```

```
param_grid3= {'n_estimators': [150, 200, 250, 300, 350],
              'max_depth': [None] }
tree_tuner1 = GridSearchCV(RandomForestClassifier(random_state=42),param_grid3,cv=10, refit=True,verbose=3 )
tree_tuner1.fit(X_pca140, np.ravel(y_train))

tree_tuner1.best_params_,  tree_tuner1.best_estimator_, tree_tuner1.best_score_
```




```
0.527 (+/-0.015) for {'max_depth': 3, 'n_estimators': 5}
0.557 (+/-0.022) for {'max_depth': 3, 'n_estimators': 10}
0.597 (+/-0.016) for {'max_depth': 3, 'n_estimators': 50}
0.605 (+/-0.013) for {'max_depth': 3, 'n_estimators': 150}
0.610 (+/-0.013) for {'max_depth': 3, 'n_estimators': 200}
0.594 (+/-0.026) for {'max_depth': 5, 'n_estimators': 5}
0.628 (+/-0.014) for {'max_depth': 5, 'n_estimators': 10}
0.671 (+/-0.015) for {'max_depth': 5, 'n_estimators': 50}
0.677 (+/-0.013) for {'max_depth': 5, 'n_estimators': 150}
0.675 (+/-0.015) for {'max_depth': 5, 'n_estimators': 200}
0.698 (+/-0.023) for {'max_depth': 10, 'n_estimators': 5}
0.741 (+/-0.020) for {'max_depth': 10, 'n_estimators': 10}
0.769 (+/-0.021) for {'max_depth': 10, 'n_estimators': 50}
0.777 (+/-0.011) for {'max_depth': 10, 'n_estimators': 150}
0.778 (+/-0.012) for {'max_depth': 10, 'n_estimators': 200}
0.739 (+/-0.014) for {'max_depth': None, 'n_estimators': 5}
0.789 (+/-0.009) for {'max_depth': None, 'n_estimators': 10}
0.838 (+/-0.011) for {'max_depth': None, 'n_estimators': 50}
0.849 (+/-0.011) for {'max_depth': None, 'n_estimators': 150}
0.850 (+/-0.011) for {'max_depth': None, 'n_estimators': 200}
```

Figure: Accuracy Vs n_estimators for Random Forest Classifier.

```
RFC time for training is 464.44170578099875 and time for test is 5.178215595002257

Accuracy: 84.73%

[[2214  141   23    0    1]
 [ 130 1993  260   36    5]
 [  28  193 1855  246   42]
 [   0   30  264 1905  168]
 [   9    8   25  223 2201]]

              precision    recall  f1-score   support

           0       0.93      0.93      0.93      2379
           1       0.84      0.82      0.83      2424
           2       0.76      0.78      0.77      2364
           3       0.79      0.80      0.80      2367
           4       0.91      0.89      0.90      2466

    accuracy                           0.85     12000
   macro avg       0.85      0.85      0.85     12000
weighted avg       0.85      0.85      0.85     12000
```

1.4 Result Analysis : Use a ROC curve used for some method in your initial or results analysis such as exploring the impact on accuracy of some parameter.

ROC and AUC ROC is a probability curve showing the probability of distinguishing whether a test data belongs to a class or not. ROC curve is plotted with True Positive Rate against False Positive Rate at various threshold settings, where[8] :

True Positive Rate (TPR) = True Positive / (True Positive + False Negative) = Recall/ Sensitivity
i.e. out of all positive cases, how many instance are identified correctly
False Positive Rate (FPR) = False Positive / (True Negative + False Positive) = (1 – Specificity)

Thus, ROC curve plots sensitivity vs (1- specificity). On the other hand, AUC is the area under the ROC curve that says the probability that a classifier will predict a data point as positive than a negative one. So, the larger the area, the better the classifier.

We know that the ROC curve is generally used in binary classification. As we had 5 classes, we had to binarize our classes. To plot ROC for 5 classes, we have used 'one vs all' methodology.

- **KNN** As we see from the ROC of KNN, the classifier does better work in predicting class '0'. With a threshold of 0.2 (steepness of the curve), it classifies class '0'. But, it performs relatively bad for class '2', the AUC is 0.86.

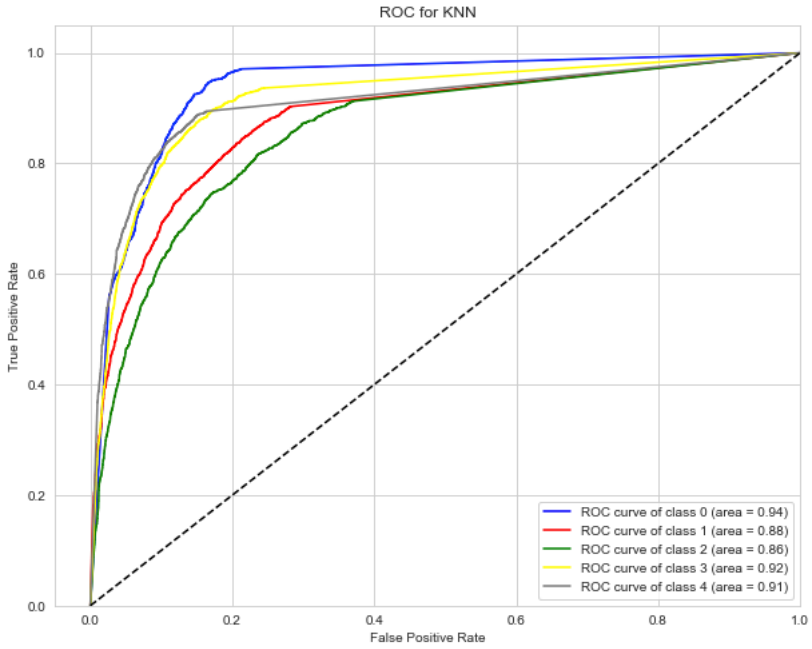


Figure: ROC-AUC for KNN.

- **SVM** ROC is based on the probabilities. But, in the SVM classifier, it does not provide a probability in scikit-learn, therefore we used decision_matrix to tune the decision. So, it is not like other classifiers. Here we notice that like KNN, SVM was good at classifying class '0', but poor at classifying class '1'.

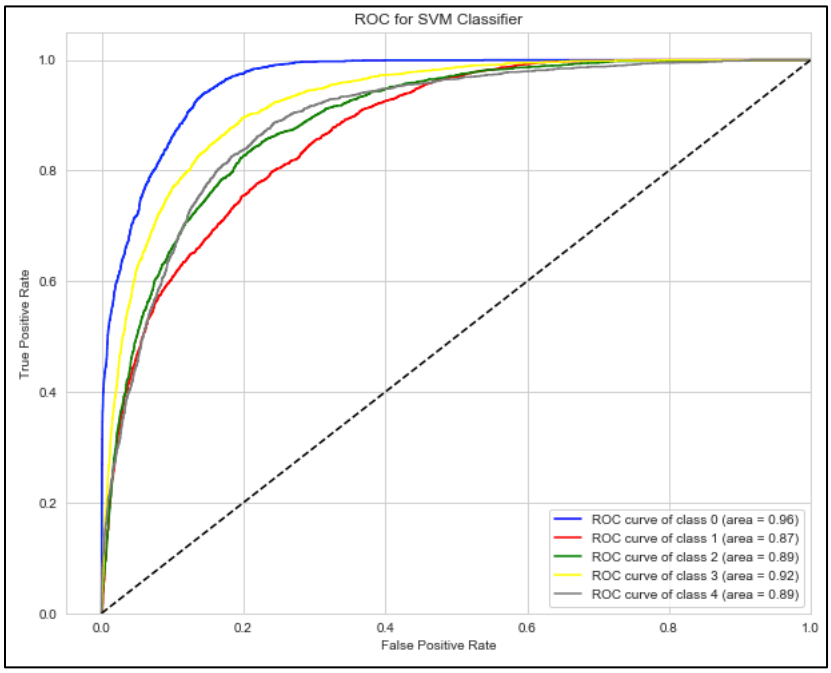


Figure: ROC-AUC for SVM.

- **Decision Tree Classifier** We can see that the performance of the decision tree classifier is bad in comparison with other classifiers. The AUC for all classes is poorer than other classifiers. If we mark the steepness of the curve, we can see that line for class '4' has a sharp rise only after it crosses the threshold at 0.9.

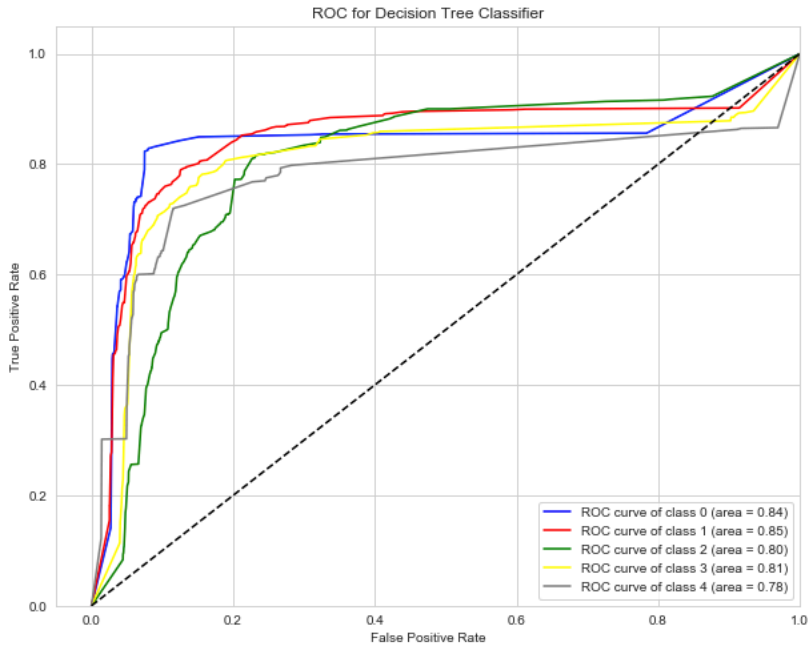


Figure: ROC-AUC for Decision Tree Classifier.

- **ROC for Random Forest Classifier** In our analysis, we found the best prediction with the Random Forest classifier, which is reflected in the ROC curve and AUC. In general, all the classifiers performed poorly in classifying class '2'. Random Forest classifier also performed badly in classifying class '2'.

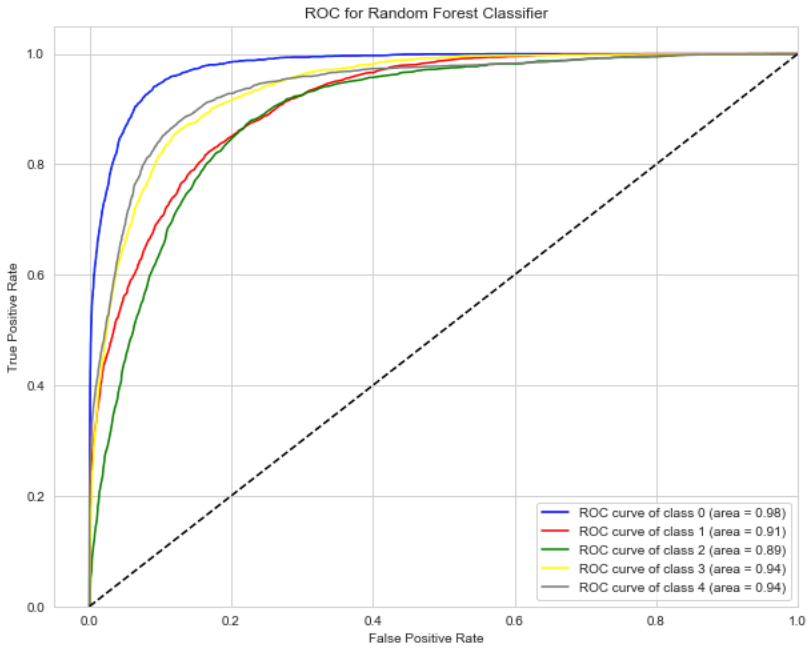


Figure: ROC-AUC for Random Forest Classifier.

1.4 Result Analysis : Evaluate your code with other metrics on the training data (by using some of it as test data) and argue for the benefit of you approach.

We have used confusion matrix to visualize the performance of our classifiers. In addition, we have used the following metrics for measuring the performance of our classifier models:

- Classification Accuracy = (True Positive + True Negative)/ (all prediction): Number of correct prediction divided by total number of predictions.
- Precision = True Positive/ (True Positive + False Negative): Out of all predicted positive instances, how many are correctly predicted.
- Recall/ Sensitivity = True Positive / (True Positive + False Negative): Out of all positive cases, how many instance are identified correctly
- F1 Score = 2* Precision * Recall/ (Precision +Recall): A measure of test's accuracy. Final score is calculated by obtaining micro-averaging or macro-averaging.
- Specificity = True Negative/ (True Negative + False Positive)
- ROC and AUC

For any classification problem, the metrics we have used are generally measured. It has found that classification accuracy provides a good estimation of the model's performance. Again, there is a trade-off between precision and recall. In general, we found that Class '2' was worst in prediction in terms of precision and recall. Detail measures are given with the classifiers.

References:

[1] "scikit-learn: machine learning in Python — scikit-learn 0.22.2 documentation." [Online]. Available: https://scikit-learn.org/stable/. [Accessed: 22-Apr-2020].

[2] A. Rosebrock, *Deep Learning for Computer Vision with Python: ImageNet Bundle*. PyImageSearch, 2017.

[3] J. Grus, *Data science from scratch: first principles with python*. O'Reilly Media, 2019.

[4] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[5] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[6] A. C. Müller, S. Guido, and others, *Introduction to machine learning with Python: a guide for data scientists*. " O'Reilly Media, Inc.," 2016.

[7] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*, vol. 112. Springer, 2013.

[8] "Understanding AUC - ROC Curve - Towards Data Science." [Online]. Available: https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5. [Accessed: 20-Apr-2020].

[9] J. Hernández-Orallo, "ROC curves for regression," *Pattern Recognit.*, vol. 46, no. 12, pp. 3395–3411, Dec. 2013.

[10] "20 Popular Machine Learning Metrics. Part 1: Classification & Regression Evaluation Metrics." [Online]. Available: https://towardsdatascience.com/20-popular-machine-learning-metrics-part-1-classification-regression-evaluation-metrics-1ca3e282a2ce. [Accessed: 15-Apr-2020].

[11] "k-Neighbors Classifier with GridSearchCV Basics - Erik G. - Medium." [Online]. Available: https://medium.com/@erikgreenj/k-neighbors-classifier-with-gridsearchcv-basics-3c445ddeb657. [Accessed: 16-Apr-2020].

[12] "Visualizing Machine Learning Models: Examples with Scikit-learn, XGB and Matplotlib." [Online]. Available: http://queirozf.com/entries/visualizing-machine-learning-models-examples-with-scikit-learn-and-matplotlib. [Accessed: 12-Apr-2020].

[13] "XGBoost Documentation — xgboost 1.1.0-SNAPSHOT documentation." [Online]. Available: https://xgboost.readthedocs.io/en/latest/. [Accessed: 20-Apr-2020].

[14] "Stack Overflow - Where Developers Learn, Share, & Build Careers." [Online]. Available: https://stackoverflow.com/. [Accessed: 22-Apr-2020].

[15] "Data Science Stack Exchange." [Online]. Available: https://datascience.stackexchange.com/. [Accessed: 22-Apr-2020].

[16] "Machine Learning - Medium." [Online]. Available: https://medium.com/topic/machine-learning.

[17] "Data Science - Medium." [Online]. Available: https://medium.com/topic/data-science. [Accessed: 16-Apr-2020].