Question 1: Classification: Feature Extraction + Classical Methods

Feature Extraction

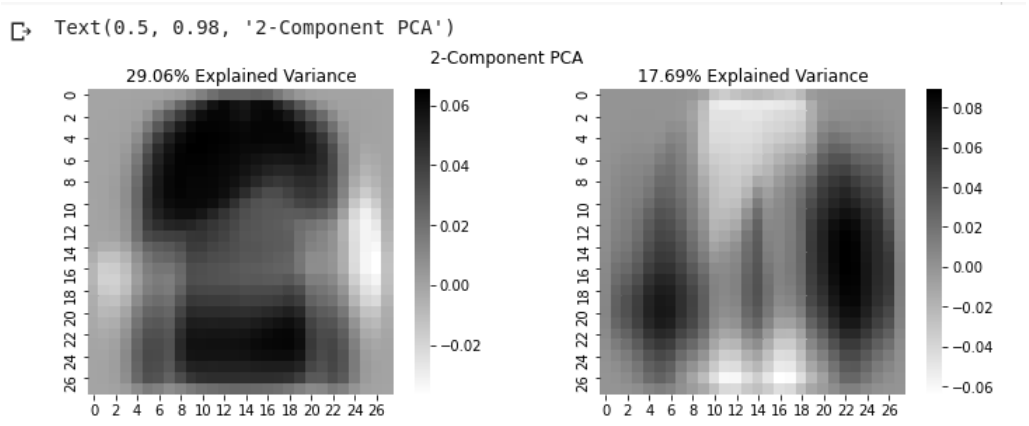
Feature extraction techniques helps in accuracy improvements, overfitting risk reduction, speed up in training, improve Data Visualization.

Experiments with PCA:

For the given Fashion MNIST data the first two components of PCA explains about 46.75 % of the total variance.

When plotted how much weight each pixel in the clothing picture gets in the resulting vector, using a heatmap. This created a interpretable picture of what each vector is "finding".

```
Text(0.5, 0.98, '2-Component PCA')
```



The first component looks like...some kind of large clothing object (e.g. not a shoe or accessor) something like a shirt. The second component looks like negative space around a pair of pants.

When using PCA, it's a good idea to pick a decomposition with a reasonably small number of variables by looking for a "cutoff" in the effectiveness of the model.

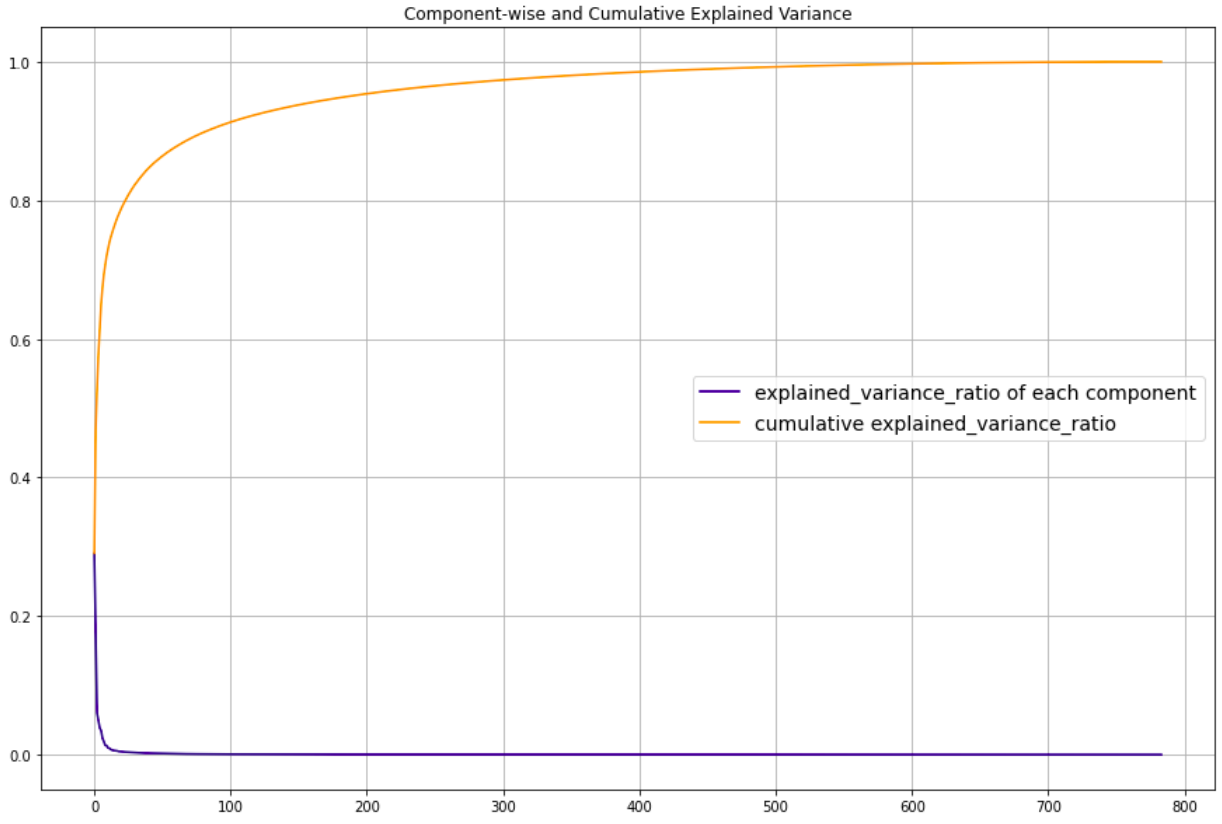
From the below plot for cumilative explained variance ratio and for explained variance ratio for each of the components of the PCA, we can observe that first components of PCA contribute to the most of the variance.

We can see that only 100 PCA components can explain more than 90% variance.Selecting just 100 components among 784 decreases the size of data largely and can result in less training time.We will look at the results of the training time in coming discussion and how it helps in avoiding overfitting the model.

code for plotting explaned variace graph

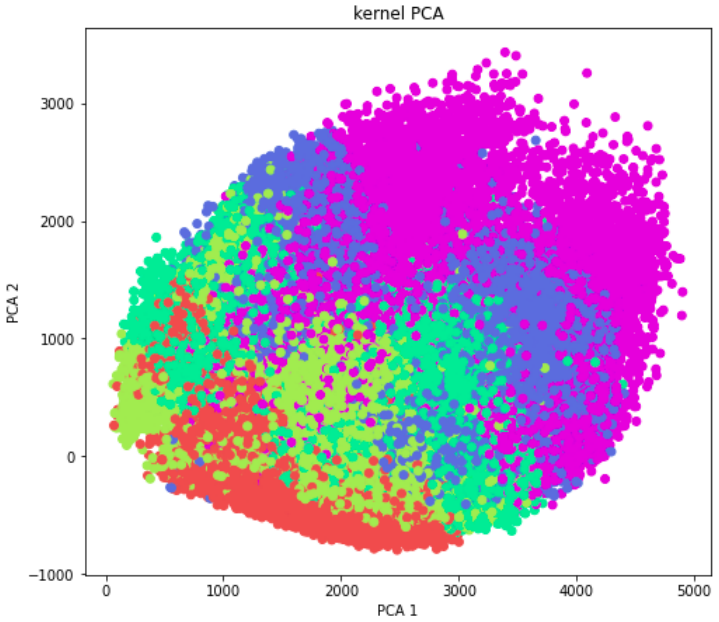
```
pca = PCA().fit(data)
plt.figure(figsize=(15,10))
plt.plot(pca.explained_variance_ratio_,color='navy',label='explained_variance_ratio of each component')
plt.plot(np.cumsum(pca.explained_variance_ratio_),color='darkorange',label='cumulative explained_variance_ratio')
plt.title("Component-wise and Cumulative Explained Variance")
plt.legend(loc="center right",prop={'size': 14})
plt.grid()
plt.show()
```

explaned variace graph

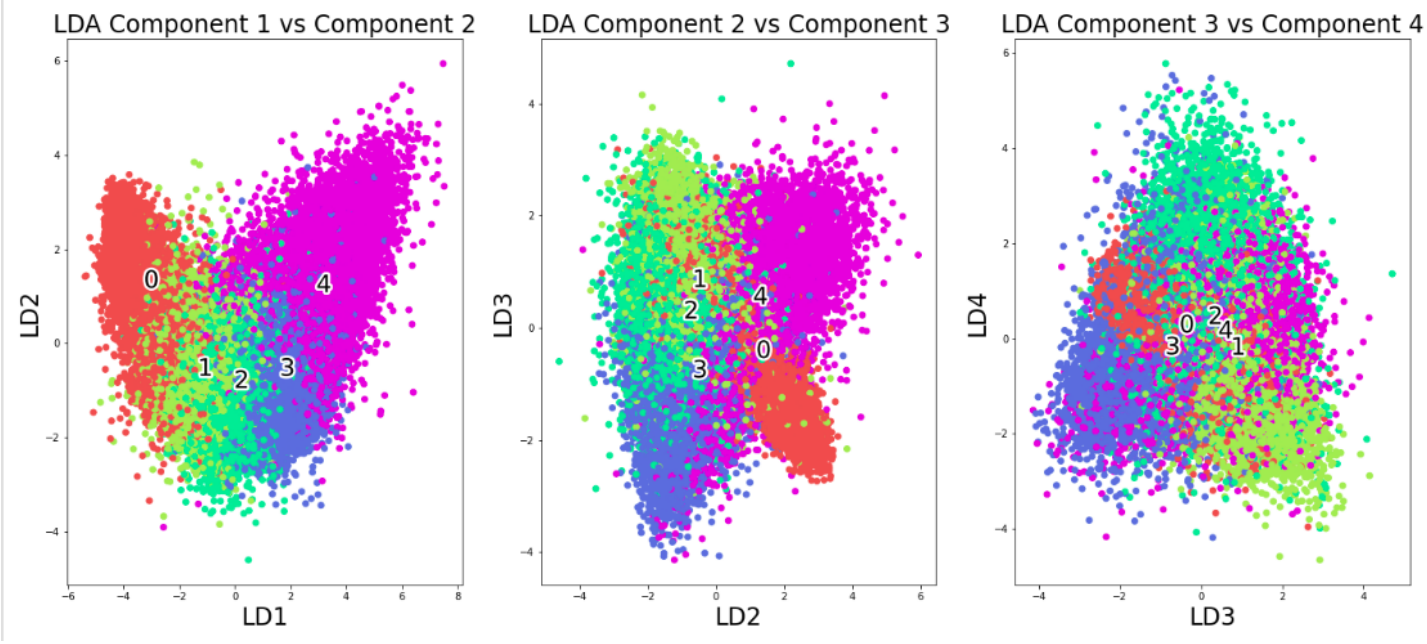


Visualising first two components of PCA

```
def scatter_plot(x, colors,method):
    # choose a color palette with seaborn.
    num_classes = len(np.unique(colors))
    palette = np.array(sns.color_palette("hls", num_classes))

    # create a scatter plot.
    f = plt.figure(figsize=(8, 8))
    ax = plt.subplot(aspect='equal')
    plt.xlabel("PCA 1")
    plt.ylabel("PCA 2")
    sc = ax.scatter(x[:,0], x[:,1], c=palette[colors.astype(np.int)])
    ax.set_title(method)

scatter_plot(data_pca[:,0:2], labels,"kernel PCA")
```



Visualization with LDA:



Exploring Classic ML models

KNN

First checking how a simple algorithm like **KNN** is performing on the dataset

We can check how much time KNN will be taking on a subset of samples(10k) with all features vs considering first 120 components of PCA.

KNN for 10k Samples with val and test sets				
features considered	**Training time(sec)**	**Prediction Time(sec)**	**Cross Val Mean Acc**	**Test accuracy**
784 raw	0.75564956	31.26071	0.7843749	0.7955
120 PCA	0.1107163	1.63198	0.8166249	0.8185

- Cross Validation mean accuracy score for C= 13 with all features is 0.7843749 where as it is 0.8166249 with 120 PCA components
- time taken for training KNN for all features is 0.75564956 sec whereas it is just 0.1107163 sec
- test set accuracy score for C= 13 with all features is 0.7955 and with 120 PCA components is 0.8185
- time taken for predicting KNN for all features is 31.26071 sec and with 1.63198 sec

We can notice that with about only 120 PCA components , KNN's accuracy for crass validation and test is better than considering all features along with less time taken for both training and predicting.As said above feature extraction helped in improving each and every aspect of KNN in this case.

So we can now start tuning KNN for different number of PCA components and with different parameters to achieve good accuracy.

Using Grid Search to find the best parameters

```
# defining the parameter values that should be searched
k_range = list(range(11,71,15))
weight_options = ['uniform', 'distance']
# intialising search grid
param_grid = dict(n_neighbors=k_range, weights=weight_options)
print(param_grid)
# splitting data
train_x,test_x,train_y,test_y=train_test_split(data_pca_subset[0:30000],labels[0:30000],test_size=0.2,random_state=42)

grid = GridSearchCV(knn, param_grid, cv=10, scoring='accuracy')
grid.fit(train_x, train_y)

print(grid.best_estimator_)
print(grid.best_score_)
print(grid.best_params_)
```

For 30,000 samples : obtained best parameters and its accuracy are :

best parameters : {'n_neighbors': 11, 'weights': 'distance'}

Accuracy : 0.8476250000000001

So lets train KNN for whole dataset with 120 PCA components and test the accuracy with a small test set.

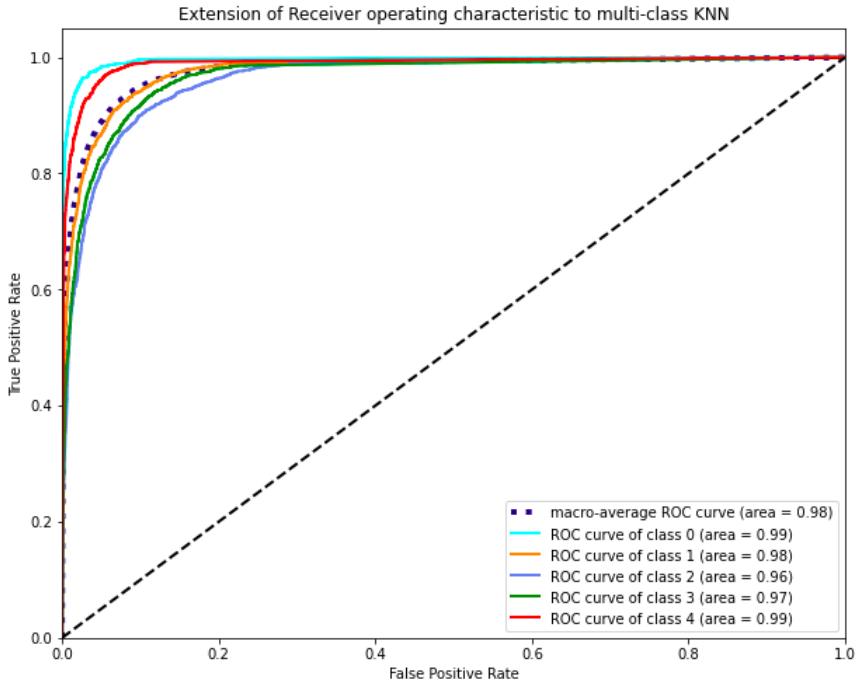
```
train_x,test_x,train_y,test_y=train_test_split(data_pca_subset,labels,test_size=0.1,random_state=42)

time_start = time.time()
knn = KNeighborsClassifier(n_neighbors=11,weights='distance')
knn.fit(train_x, train_y)
time_end = time.time()
scores=cross_val_score(knn, train_x ,train_y, cv=10)

print("Validation set mean accuracy score for C= {0} with 120 PCA components is {1}".format(11,scores.mean()))
print("time taken for training KNN with 120 PCA components {0}".format(time_end -time_start))

# testing how much time KNN will take the predict the output
time_start = time.time()
y_pred = knn.predict(test_x)
time_end = time.time()
accscore = accuracy_score(test_y, y_pred)
print("test set accuracy score for C= {0} with 120 PCA components is {1}".format(13,accscore))
print("time taken for predicting output with 120 PCA components {0}".format(time_end -time_start))
```

- Validation set mean accuracy score for C= 11 with 120 PCA components is **0.8617**
- Time taken for training KNN with 120 PCA components **1.1164 sec**
- Test set accuracy score for C= 11 with 120 PCA components is **0.863**
- Time taken for predicting output with 120 PCA components **24.1985 sec**

KNN is a Lazy learning algorithm.Lazy learning is a learning method in which generalization of the training data is, delayed until a query is made to the system.So to predict the results KNN has to caluclate distances for multiple data points.As a result predicting results in KNN take huge time when compared to training.

If predicting quickly is our application,then KNN is not our guy.

KNN ROC : ROC graph for different class of KNN along with a macro-average curve.



- We can see that AuC under macro-average curve is .98.
- The classification performed by the model is less than average for 'class 1' and 'class 2'
- This model seems to classify 'class 0' and 'class 4' to good extent when compared to other class.

Lets train KNN with all data sample without any training set and then predict the labels for test set and then submit the predictions in Kaggle.

- **time taken for training KNN with 120 PCA components 0.972000 sec**
- **time taken for prediction using KNN with 120 PCA components 59.21179 sec**
- **Kaagle Submission Accuracy :0.85780**

Lets try bit more complex model like Random Forest

Random Forest

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that work by building a multitude of decision trees at training time and outputting the class which is the class mode (classification) or mean prediction (regression) of the individual trees.

Grid search Results for Random Forest for 5000 samples is as shown below

```
Mean Accuray score when depth=20 and num of tress=50 is : 0.762
Mean Accuray score when depth=20 and num of tress=150 is : 0.786
Mean Accuray score when depth=20 and num of tress=200 is : 0.7830000000000001
Mean Accuray score when depth=20 and num of tress=300 is : 0.7847500000000001
Mean Accuray score when depth=30 and num of tress=50 is : 0.766
Mean Accuray score when depth=30 and num of tress=150 is : 0.782
Mean Accuray score when depth=30 and num of tress=200 is : 0.7855
Mean Accuray score when depth=30 and num of tress=300 is : 0.7845
Mean Accuray score when depth=50 and num of tress=50 is : 0.7660000000000001
Mean Accuray score when depth=50 and num of tress=150 is : 0.7825
```

```
Mean Accuray score when depth=50 and num of tress=200 is : 0.786
Mean Accuray score when depth=50 and num of tress=300 is : 0.78375
Mean Accuray score when depth=60 and num of tress=50 is : 0.7660000000000001
Mean Accuray score when depth=60 and num of tress=150 is : 0.7825
Mean Accuray score when depth=60 and num of tress=200 is : 0.786
Mean Accuray score when depth=60 and num of tress=300 is : 0.78175
Mean Accuray score when depth=None and num of tress=50 is : 0.7660000000000001
Mean Accuray score when depth=None and num of tress=150 is : 0.7825
Mean Accuray score when depth=None and num of tress=200 is : 0.786
Mean Accuray score when depth=None and num of tress=300 is : 0.78075
```

We can observer that for **depth =20 and trees = 150** ,mean validation accuracy seems to be good.

Lets analysie it using ROC curve for (depth =20,trees = 10) ,(depth =20,trees = 20) and (depth =20,trees = 150)

Considering the extension of ROC for multiclass variables by considering macro-average of the ROC curves of each class.

Reason to consider Macro-average instead of micro-average: Micro- and macro-averages will calculate slightly different things, and thus differ in their interpretation. A macro-average calculates the metric for each class independently and then takes the average (thus treating all classes equally) whereas a micro-average aggregates the contributions of all classes to calculate the average metric. In a multi-class classification setup, micro-average is preferable if you suspect there might be class imbalance.

In Fashion-MNIST dataset all the classes have equal and no imbalance is obsserved. So choosing macro-average over micro-average.

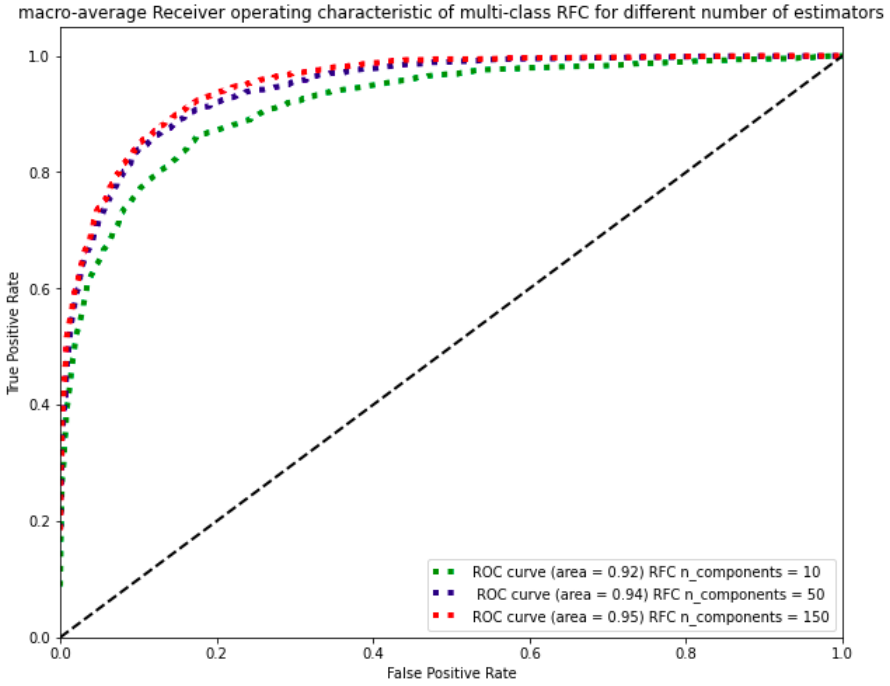
Sample code for generating macro- average ROC for each model:

```
classifier = OneVsRestClassifier(RandomForestClassifier(max_depth=20, random_state=42,n_estimators=10))
y_score = classifier.fit(X_train, y_train).predict_proba(X_test)

# Compute ROC curve and ROC area for each class
fpr_rfc_10 = dict()
tpr_rfc_10 = dict()
roc_auc_rfc_10 = dict()
for i in range(n_classes):
    fpr_rfc_10[i], tpr_rfc_10[i], _ = roc_curve(y_test[:, i], y_score[:, i])
    roc_auc_rfc_10[i] = auc(fpr_rfc_10[i], tpr_rfc_10[i])

# Compute micro-average ROC curve and ROC area
fpr_rfc_10["micro"], tpr_rfc_10["micro"], _ = roc_curve(y_test.ravel(), y_score.ravel())
roc_auc_rfc_10["micro"] = auc(fpr_rfc_10["micro"], tpr_rfc_10["micro"])

# Compute ROC curve and ROC area for each class
fpr_rfc_10 = dict()
tpr_rfc_10 = dict()
roc_auc_rfc_10 = dict()
for i in range(n_classes):
    fpr_rfc_10[i], tpr_rfc_10[i], _ = roc_curve(y_test[:, i], y_score[:, i])
    roc_auc_rfc_10[i] = auc(fpr_rfc_10[i], tpr_rfc_10[i])

# Compute micro-average ROC curve and ROC area
fpr_rfc_10["micro"], tpr_rfc_10["micro"], _ = roc_curve(y_test.ravel(), y_score.ravel())
roc_auc_rfc_10["micro"] = auc(fpr_rfc_10["micro"], tpr_rfc_10["micro"])
```

Plotting ROC graph

```
# Ploting only macro ROC curves
plt.figure(figsize=(10,8))

plt.plot(fpr_rfc_10["macro"], tpr_rfc_10["macro"],
         label='ROC curve (area = {0:0.2f}) RFC n_components = 10'
```

```
              ''.format(roc_auc_rfc_10["macro"]),
         color='green', linestyle=':', linewidth=4)

plt.plot(fpr_rfc_50["macro"], tpr_rfc_50["macro"],
         label=' ROC curve (area = {0:0.2f}) RFC n_components = 50'
              ''.format(roc_auc_rfc_50["macro"]),
         color='navy', linestyle=':', linewidth=4)

plt.plot(fpr_rfc_150["macro"], tpr_rfc_150["macro"],
         label='ROC curve (area = {0:0.2f}) RFC n_components = 150'
              ''.format(roc_auc_rfc_150["macro"]),
         color='red', linestyle=':', linewidth=4)

lw = 2
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('macro-average Receiver operating characteristic of multi-class RFC for different number of estimators')
plt.legend(loc="lower right")
plt.show()
```



We can observe that by changing the number of components for RFC the AUC under ROC's is changing accordingly.

From the graph we can see that AUC for 150 components is greater than the remaining two ,incidating high true positive probability.So we can select the parameters for which AUC is more.

So , selecting (depth =20,trees = 150) as the best set of parameters.

Now training RFC with 120 PCA components for all samples and submitting in kaggle.

- **time taken for training RFC with 120 PCA components 122.430294 sec**
- **time taken for prediction using RFC with 120 PCA components 0.3924 sec**
- **Kaagle Submission Accuracy :0.84980**

Now training RFC with 40 PCA components for all samples and submitting in kaggle.

- **time taken for training RFC with 40 PCA components 67.3108 sec**
- **time taken for prediction using RFC with 40 PCA components 0.3455 sec**

- **Kaagle Submission Accuracy :0.87540**

When considering 120 components the model is getting overfit as we can see that the accuracy is dropping as we use more components.

We can see that the prediction time for for RFC is very less compared to its tarining time. Which is inverse of KNN.

SVM

- Support-vector machines are supervised learning models with associated learning algorithms which analyse the data used for classification and regression analysis.
- An SVM training algorithm builds a model that assigns new examples to one or the other category, given a set of training examples, each marked as belonging to one or the other category.
- An SVM model is a representation of the examples as space points, mapped in such a way that the samples of the separate categories are divided by a clear gap as large as possible.

Design Choices :

Kernel: Let us make a trail to check which kernel is performing better with 5k samples.

- Checking performance for c_Values = [0.1,1,10,20] for 5k samples and measuring the time taken for finishing scross validation.

Kernel = rbf

sample code

```
c_Values = [0.1,1,10,20]

data_pca_subset = data_pca[:,0:40]

time_start = time.time()
for x in c_Values :
    rbf_svc = svm.SVC(kernel='rbf',C=x)
    scores=cross_val_score(rbf_svc, data_pca_subset[0:5000], labels[0:5000], cv=10)
    print("Kernal = rbf and Validation set mean accuracy score for C= {0} is {1}".format(x,scores.mean()))
time_end = time.time()
print("time taken for getting cross validation score for rbf SVM with 40 PCA components {0}".format(time_end -time_start))
```

obtained rsults

```
Kernal = rbf and Validation set mean accuracy score for C= 0.1 is 0.7412
Kernal = rbf and Validation set mean accuracy score for C= 1 is 0.8208
Kernal = rbf and Validation set mean accuracy score for C= 10 is 0.8390000000000001
Kernal = rbf and Validation set mean accuracy score for C= 20 is 0.8378
time taken for getting cross validation score for rbf SVM with 40 PCA components 38.98252844810486
```

Kernel = linear

sample code

```
c_Values = [0.1,1,10,20]

data_pca_subset = data_pca[:,0:40]
time_start = time.time()
for x in c_Values :
    lin_svc = svm.SVC(kernel='linear',C=x)
    scores=cross_val_score(lin_svc, data_pca_subset[0:5000], labels[0:5000], cv=10)
    print("Kernal = rbf and Validation set mean accuracy score for C= {0} is {1}".format(x,scores.mean()))
time_end = time.time()
print("time taken for getting cross validation score for linear SVM with 40 PCA components {0}".format(time_end -time_start))
```

obtained results

```
Kernal = rbf and Validation set mean accuracy score for C= 0.1 is 0.7091999999999998
Kernal = rbf and Validation set mean accuracy score for C= 1 is 0.7086
Kernal = rbf and Validation set mean accuracy score for C= 10 is 0.7091999999999998
Kernal = rbf and Validation set mean accuracy score for C= 20 is 0.7087999999999999
time taken for getting cross validation score for linear SVM with 40 PCA components 1916.3160934448242
```

observations:

- We can see that rbf kernel's performance is better than linear for the samle data both in terms accuracy and time taken for cross validation.

So, using rbf kernel.

Trying to tune the parameters :

Results obtained for various C values and its mean cross validation accuracy s as shown below:

sample code

```
c_Values = [0.1,1,10,20,40,60,100,120]

for x in c_Values :
    lin_svc = svm.SVC(kernel='rbf',C=x)
    scores=cross_val_score(lin_svc, data_pca_subset[0:20000], labels[0:20000], cv=10)
    print("Validation set mean accuracy score for C= {0} is {1}".format(x,scores.mean()))
```

obtained results

```
Validation set mean accuracy score for C= 0.1 is 0.7952999999999999
Validation set mean accuracy score for C= 1 is 0.85185
Validation set mean accuracy score for C= 10 is 0.8676999999999999
Validation set mean accuracy score for C= 20 is 0.8693500000000001
Validation set mean accuracy score for C= 40 is 0.86295
Validation set mean accuracy score for C= 60 is 0.86
Validation set mean accuracy score for C= 100 is 0.8564499999999999
Validation set mean accuracy score for C= 120 is 0.8551
```

Now training SVM with C = 20 with 120 PCA components for all samples and testing the accuracy in kaagle.

- time taken for training SVM with 120 PCA components **329.72028 sec**
- time taken for predicting output with RFC and 120 PCA components **36.33985 sec**
- Kaagle Submission Accuracy :**0.88720**

Now training SVM with C = 20 with 50 PCA components for all samples and testing the accuracy in kaagle.

- time taken for training SVM with 50 PCA components **119.66347 sec**
- time taken for predicting output with SVM and 50 PCA components **17.000176 sec**
- Kaagle Submission Accuracy :**0.89500**

sample code

```
time_start = time.time()
lin_svc = svm.SVC(kernel='rbf',C=20).fit(pca_data_new,labels)
time_end = time.time()
print("time taken for training SVM with 50 PCA components {0}".format(time_end -time_start))

time_start = time.time()
predicted = lin_svc.predict(test_pca_new)
time_end = time.time()
print(predicted.shape)
print("time taken for predicting output with SVM and 50 PCA components {0}".format(time_end -time_start))
```

Now training SVM with C = 20 with 35 PCA components for all samples and testing the accuracy in kaagle.

- time taken for training SVM with 35 PCA components **88.02530288 sec**
- time taken for predicting output with SVM and 35 PCA components **12.8491978 sec**
- Kaagle SUbmission Accuracy :**0.89200**

Now training SVM with C = 20 with all PCA components for all samples and testing the accuracy in kaagle.

- time taken for training SVM with all PCA components **3438.19957 sec**
- time taken for predicting output with SVM and all PCA components **319.45426**
- Kaagle SUbmission Accuracy :**0.88480**

Comparing results:

SVM C value	No.of PCA components	Kaggle Accuracy	Training Time(Sec)	Prediction Time(sec)
20	35	89.2	88	13
20	50	89.5	120	17
20	120	88.72	330	36
20	784	88.48	3438	319
10	40	89	117	17

Conclusions:

From the above results we can notice that as we use more number of PCA components the model is getting overfit and the accuracy is getting decreased.

With less PCA components both training the model and prediction is quite fast without compramising much on the accuracy.

So choosing minimal PCA components can avoid overfitting the model.

Trying XGBoost

Understanding XGBoost

- With a regular machine learning model, we would simply train a single model on our dataset, like a decision tree, and use that for prediction.
- On the other hand, boosting is taking a more iterative approach. It is still technically an ensemble technique that combines many models to perform the final one together, but takes a cleverer approach.
- Instead of training all models in isolation from each other, train models are boosted successively, with each new model being trained to correct previous errors.
- XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework.
- XGBoost improves upon the base GBM framework through systems optimization and algorithmic enhancements.

Ref : XGBoost Algorithm: Long May She Reign!

Ref : Exploring XGBoost

Tried to implement XGBoost with below parameters :

Choosen an early stopping = 50 to not overfit the model.

sample code depth = 20 and n_classes = 10

```
param_list = [("eta", 0.08),  ("max_depth",20),("subsample", 0.8), ("colsample_bytree", 0.8), ("objective", "multi:softmax"), ("eval_
n_rounds = 400
early_stopping = 50

d_train = xgb.DMatrix(X_train, label=y_train)
d_val = xgb.DMatrix(X_valid, label=y_valid)
eval_list = [(d_train, "train"), (d_val, "validation")]
bst = xgb.train(param_list, d_train, n_rounds, evals=eval_list, early_stopping_rounds=early_stopping, verbose_eval=True)
```

```
param_list = [("eta", 0.08),  ("max_depth",10),("subsample", 0.8), ("colsample_bytree", 0.8), ("objective", "multi:softmax"), ("eval_
n_rounds = 400
early_stopping = 50

d_train = xgb.DMatrix(X_train, label=y_train)
d_val = xgb.DMatrix(X_valid, label=y_valid)
eval_list = [(d_train, "train"), (d_val, "validation")]
bst = xgb.train(param_list, d_train, n_rounds, evals=eval_list, early_stopping_rounds=early_stopping, verbose_eval=True)
```

Obtained an accuracy of 0.8700 in kaggle

sample code depth = 10 and n_classes = 10

```
param_list = [("eta", 0.08),  ("max_depth",10),("subsample", 0.8), ("colsample_bytree", 0.8), ("objective", "multi:softmax"), ("eval_
n_rounds = 400
early_stopping = 50

d_train = xgb.DMatrix(data_pca_subset[:,0:35], label=labels)
eval_list = [(d_train, "train")]
time_start = time.time()
bst = xgb.train(param_list, d_train, n_rounds, evals=eval_list, early_stopping_rounds=early_stopping,verbose_eval=True)
time_end = time.time()
```

Obtained an accuracy of 0.88700 in kaggle

time taken for training XGBoost with 50 PCA components is 906.3 sec ,n_rounds = 400,early_stopping = 50

time taken for predicting with XGBoost trained with 50 PCA components 2.468 sec,n_rounds = 400,early_stopping = 50

sample code depth = 10 and n_classes = 10 and 35 PCA components

Obtained an accuracy of 0.89360 in kaggle

time taken for training XGBoost with 35 PCA components is 602.01569843292236328 ,n_rounds = 400,early_stopping = 50

time taken for predicting with XGBoost trained with 25 PCA components 2.6807 sec ,n_rounds = 400,early_stopping = 50

Metrics

Now lets look at some of the Metrics observed when training and testing for Random Forest models:

max_depth	n_estimators	mean validation score	F1 Score	Log Loss
20	50	0.67125	0.6756721714	0.984716929
20	150	0.71625	0.7290259456	0.9516188218
20	200	0.718	0.7286997009	0.9636384884
20	300	0.6975	0.7207902849	0.9832752139
30	50	0.67125	0.6761274585	0.9845265709
30	150	0.69375	0.724504466	0.9706335117
30	200	0.7025	0.7316171679	0.9634077936
30	300	0.705	0.7287902849	0.9542368958
50	50	0.67125	0.6761274585	0.9845265709
50	150	0.69375	0.724504466	0.9706335117
50	200	0.7025	0.7316171679	0.9634077936
50	300	0.705	0.7287902849	0.9542368958

- We can see that mean validation score is highest for parameters max_depth = 20 and number of estimators = 200.
- If we observe the highest F1 Score and lowest Log Loss values are associated with max_depth = 20 and number of estimators = 150.

- If we have considered only mean validation score to finalise the parameters , we would have choosen max_depth = 20 and number of estimators = 200 which has more Log loss nad less F1 score compared to above prameters results.
- So splitting training data into train and test sets and checking the model on test samples with metrics like Log Loss,F1 score,confusion matrix help in undersatnding how the model is performing.
- Taking into consideration only mean validation accuracy and selecting model might result in ending with a overfit model.

So the selected parameters were max_depth = 20 and n _estimators = 150.

sample code used

```
for x in depth :
    for y in trees:
        clf=RandomForestClassifier(max_depth=x, random_state=42,n_estimators=y)
        scores=cross_val_score(clf, train_x, train_y, cv=10)
        clf.fit(train_x, train_y)
        print("Mean Accuray score when depth={0} and num of tress={1} is : {2}".format(x,y,scores.mean()))
        predictions = clf.predict(test_x)
        f1_s = f1_score(test_y, predictions,average='macro')
        f1score.append(f1_s)
        print("F1 score when depth = {0} and trees = {1} is {2}".format(x,y,f1_s))
        rfcpred = clf.predict_proba(test_x)
        lg_loss = log_loss(test_y, rfcpred)
        logloss.append("Logarithm loss when depth ="+ str(x) + " and trees = " + str(y) + " is : " + str(lg_loss))
        print("Logarithm loss when depth = {0} and trees = {1} is {2}".format(x,y,lg_loss))
        print(confusion_matrix(test_y,predictions))
        deatils.append([x,y,scores.mean(),f1_s,lg_loss,confusion_matrix(test_y,predictions)])
```

Comparision:

Lets compare the results of different models tried above:

Model	model description	PCA Components	training time (sec)	prediction time(sec)	kaggle accuracy
KNN	C=11	120	0.972	59.21	85.78
Random Forest	max_depth = 20 n_estimators = 150	120	122.43	0.392	84.98
Random Forest	max_depth = 20 n_estimators = 150	40	67.31	0.3455	87.54
SVM	kernal = rbf and C = 10	40	117	17	89
SVM	kernal = rbf and C = 20	35	88	13	89.74
SVM	kernal = rbf and C = 20	50	120	17	89.5
SVM	kernal = rbf and C = 20	120	330	36	88.72
XGBoost	depth = 10 n_classes=10	50	906.3	2.46	88.7
XGBoost	depth = 10 n_classes=10	35	602.3	2.68	89.36

points to be noted

- We can notice that Randform forest is the quickiest in terms of prediction followed by XGBoost and SVM.Knn is very slow in predicting/classification.
- As tree based classifiers build tress while training, prediction is very quick as they have to traverse through the tree.It explains the speed of Random Forests and XGBoost in prediction.
- The more PCA components we use ,the more model is getting overfit.So using only required number of PCA components is a key in model tuning in all aspects.

Highest Kaggle Score recieved using Classic ML models is 89.76%

Team Name : The Beard guy