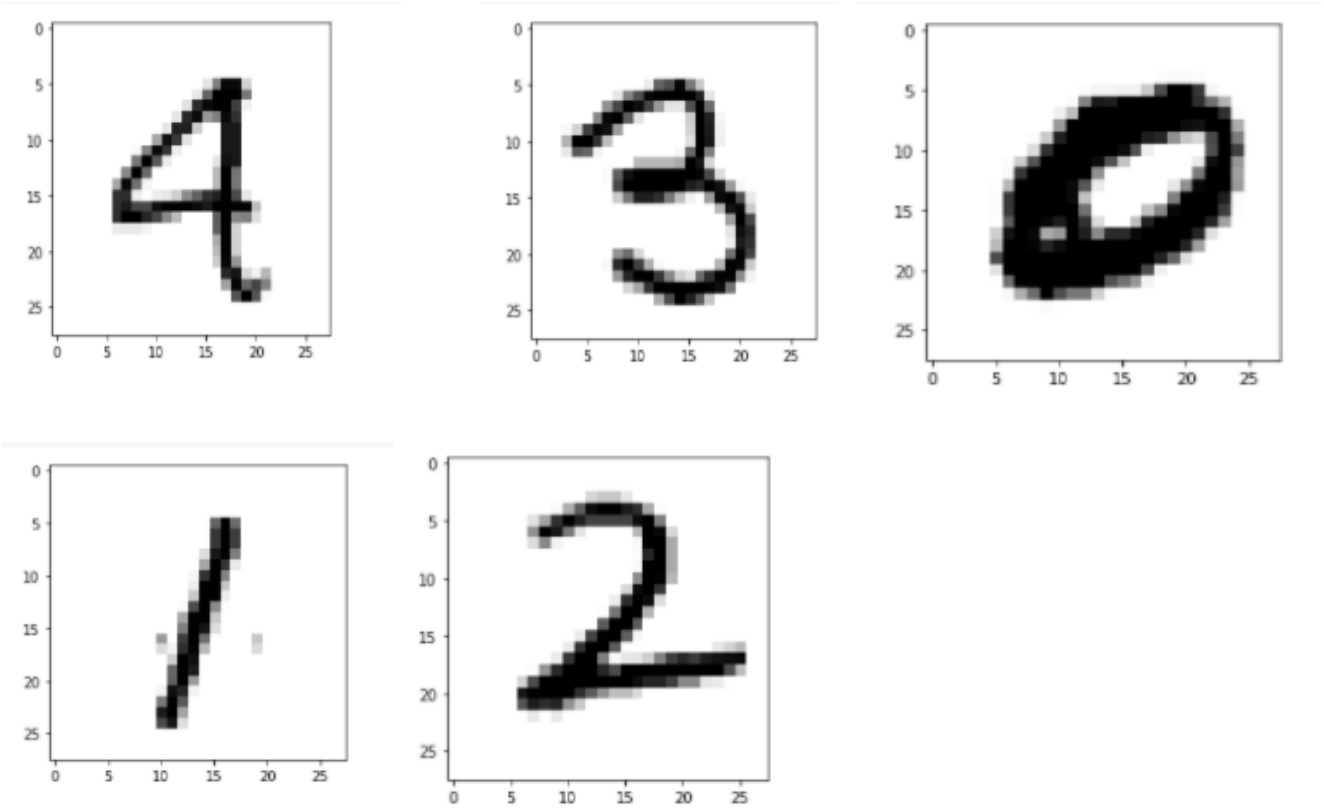
1.

In this question , we need to work with MNIST data and use the training data to develop a model which will take an input image and classify it into one of the 5 classes .



1. Use 70-20-10 split for training, validation and testing. :

Firstly we fetch the MNIST data from sklearn using :

```
X_mn, y_mn = fetch_openml('mnist_784', version=1,
return_X_y=True)
```

The output of the code are 2 numpy arrays consisting of feature values and class values respectively for every data sample.

We convert the numpy arrays to pandas dataframe and convert the string type of class labels to int type.

```
df_mn = pd.DataFrame(X_mn, columns = [i for i in range
(784)])
```

To split the data into training , validation and testing sets , we will use sklearn's train-test-split 2 times.

```
X_train, X_test, y_train, y_test =
train_test_split(x_mn_n, y_mn_n, test_size=0.1,
random_state=1)
X_train, X_val, y_train, y_val =
train_test_split(X_train, y_train, test_size=0.22222,
random_state=1)
```

2. Use nearest neighbour, perceptron and SVM classifiers

For hyperparameter tuning , we need to grid search for a specific set of parameters.

'Grid Search applied for **'n_neighbors'** tuning in knn:

```
clf_k =  KNeighborsClassifier()
parameters = {'n_neighbors':[15,20]}
clf_k_fin = GridSearchCV(clf_k, parameters)
clf_k_fin.fit(X_train, y_train)
accu_k = clf_k_fin.score(X_test)
```

The best value of 'n_neighbours' comes out to be 20

On fitting this best classifier on train data , we get accuracy on test data as :

```
0.9865696698377169
```

Grid search applied for tuning **‘max_iter’** in MLP classifier :

```
clf_mlp = MLPClassifier()
parameters = {'learning_rate':[0.001, 0.01, 0.1],
'max_iter':[200,300,400]}
clf_mlp_fin= GridSearchCV(clf_mlp, parameters)
accu_mlp = clf_mlp_fin.score(X_test)
```

The best value of ‘max_iter’ comes out to be 100 . On fitting this best classifier on train data , we get an accuracy on test data as:

```
0.9868494683827644
```

Similarly Using grid search for tuning ‘C’ value in SVM model :
The best value of C comes out to be 5 .
Accuracy obtained on test data :

```
0.9941242305540011
```

Models	KNN	MLP	SVM
Accuracy	0.98656966983	0.98684946838	0.9941242305

3. Normalize the data by mean subtraction followed by standard deviation division.

Normalising the train, test and validation data :

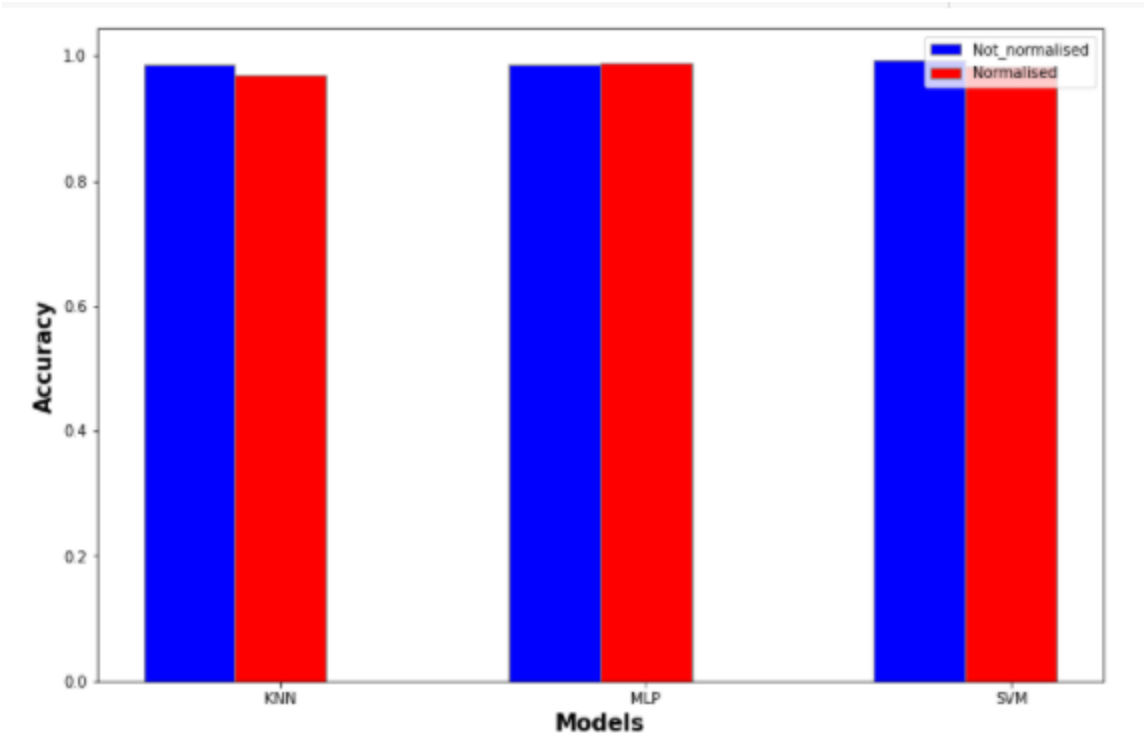
```
scaler = StandardScaler()
scaler.fit(X_train)
X_train= scaler.transform(X_train)
scaler.fit(X_val)
X_val= scaler.transform(X_val)
scaler.fit(X_test)
X_test = scaler.transform(X_test)
```

And applying the same steps as in above part :

We get

Models	KNN	MLP	SVM
Accuracy(normalised)	0.97006155	0.9879686	0.98265249

Comparing accuracies for 2nd and 3rd part :



4. Implement any two from OVA/OVO/DAG, :

From the above parts , we found that SVM gives the highest accuracy on test data .

Applying OVA :

```
clf_ova_svm =
OneVsRestClassifier(svm.SVC()).fit(X_train,
y_train)
accu_ova_svm = clf_ova_svm.score(X_test, y_test)
```

Applying OVO :

```
clf_ovo_svm = clf =
OneVsOneClassifier(svm.SVC()).fit(X_train,
y_train)
```

```
accu_ovo_svm = clf_ovo_svm.score(X_test, y_test)
```

	OVA	OVO
Accuracy	0.979854	0.99244

2.

Given the diabetes dataset , we need to perform the required steps of operation.

1. **SVM classifier (using a linear kernel):**
   Before applying svm , we need to standardize the data which can be done by using :

```
scaler = StandardScaler()
scaler.fit(X)
X= scaler.transform(X)
```

where X is the input data after dropping the ‘Outcome’ column .

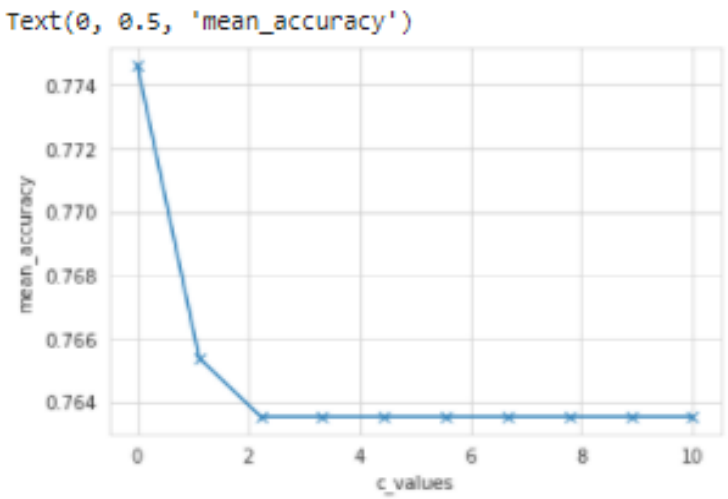
To get performance on various classifiers , we will divide the data into a train-validation-test set using the same code from question 1.

Applying SVM classifier with linear kernel :

```
clf = svm.SVC(kernel='linear')
clf.fit(X_train, y_train)
clf.score(X_val, y_val)
clf.score(X_test, y_test)
```

Trying different values of C for SVM and plotting scores corresponding to each C values :

```
c_values = np.linspace(0.01, 10, 10)
scores =
[cross_val_score(clf.set_params(C=c_value),X_train
,y_train,cv=5) for c_value in c_values]
means = np.array([np.mean(score) for score in
scores])
deviations = np.array([np.std(score) for score in
scores])
```



So we get maximum mean_accuracy for C= 0.01

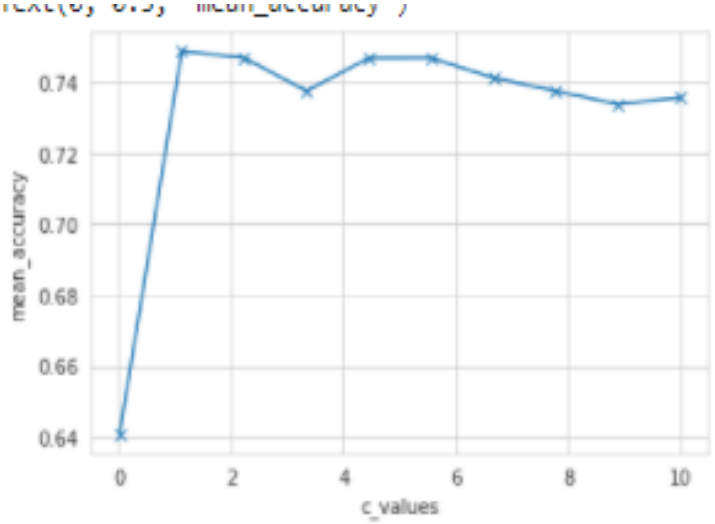
2. SVM classifier (using a Polynomial kernel and a Gaussian kernel)

Polynomial kernel :

```
clf_poly = svm.SVC(kernel='poly')
clf_poly.fit(X_train , y_train)
```

```
clf_poly.score(X_val, y_val)
clf_poly.score(X_test, y_test)
```

Trying different values of C for SVM with polynomial kernel, we get the following:

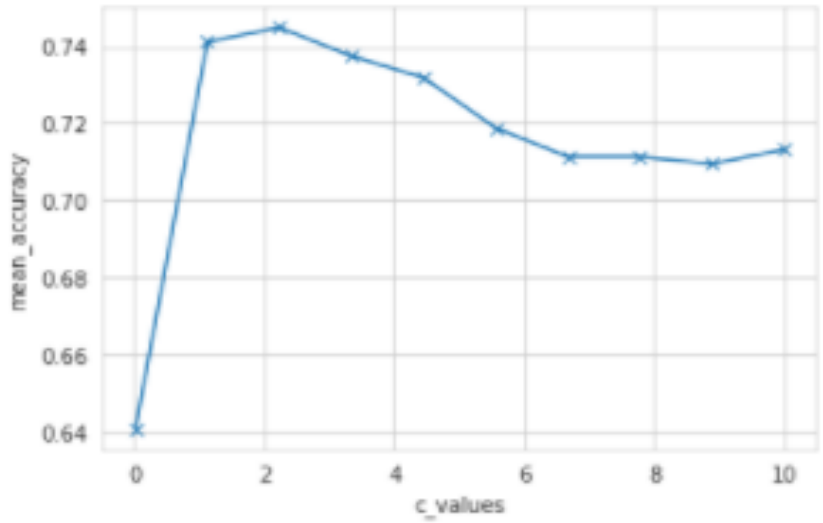


So we get maximum mean accuracy for C= `1.12`

Gaussian kernel :

```
clf_gauss = svm.SVC(kernel='rbf')
clf_gauss.fit(X_train , y_train)
clf_gauss.score(X_val, y_val)
clf_gauss.score(X_test, y_test)
```

Trying different C values , we get :



Maximum accuracy is for C= `2.23`

Comparing linear, polynomial and gaussian kernels for SVM classifier

	Linear	Polynomial	Gaussian
Accuracy(test data)	`0.79220`	`0.74025`	`0.8181`

Therefore we get maximum accuracy for the Gaussian kernel .

3. Report the number of support vectors obtained in the final model in each case.

We can find the number of support vectors obtained in final model for each case using :

For linear kernel :

```
clf_lin_fin = svm.SVC(kernel= 'linear', C=0.01)
clf_lin_fin.fit(X_train , y_train)
Clf_lin_fin.n_support_
```

Note that we are using those values of c which we found in last parts.
Similar procedure is done for Gaussian and Polynomial kernel :

	Linear	Polynomial	Gaussian
Support vectors	[172, 170]	[155, 150]	[156, 146]

4. Perform an experiment to visualize the separating hyper-plane:

To visualise the separating hyperplane , we will first apply PCA to the data and then fit the svm model on the data .

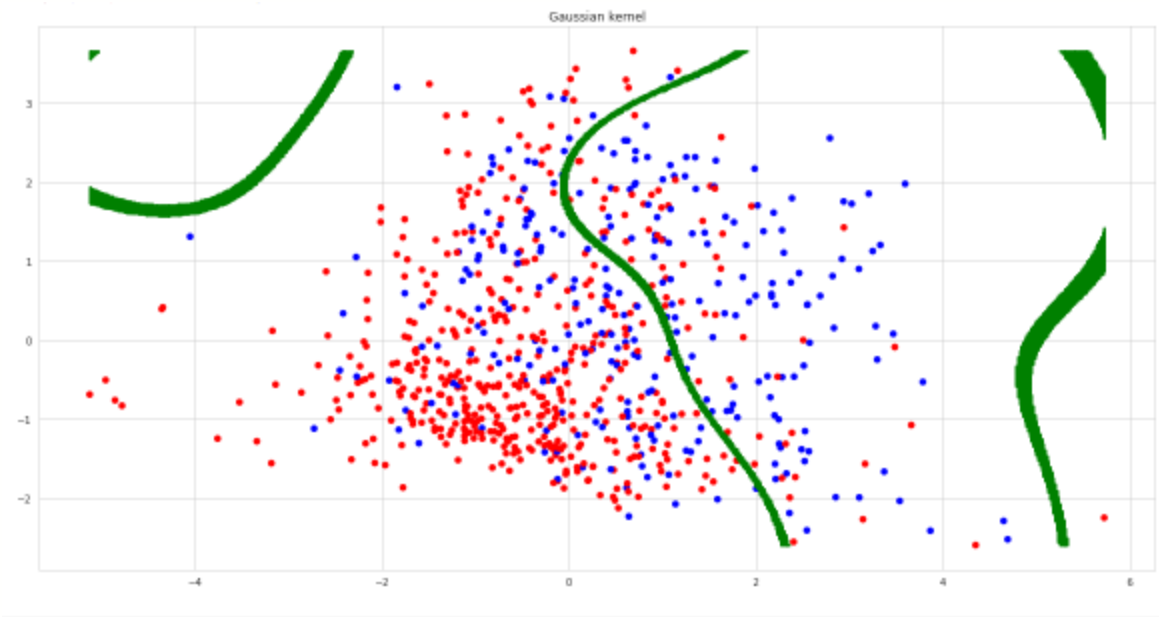
Applying PCA :

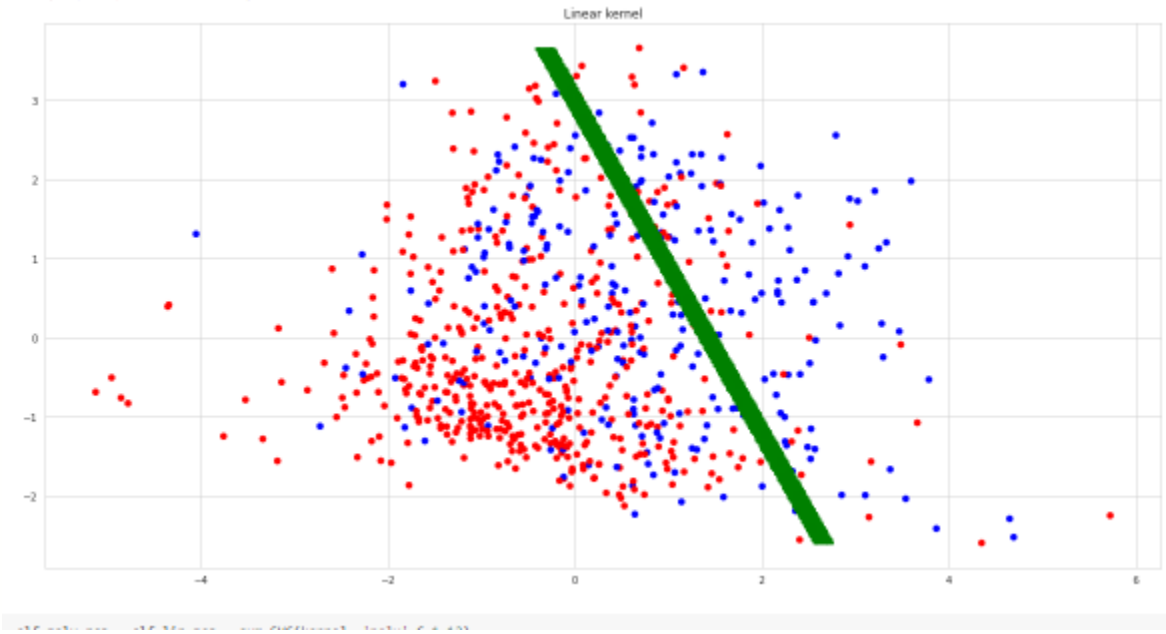
```
pca = PCA(n_components=2)
pca.fit(X)
X= pca.transform(X)
```

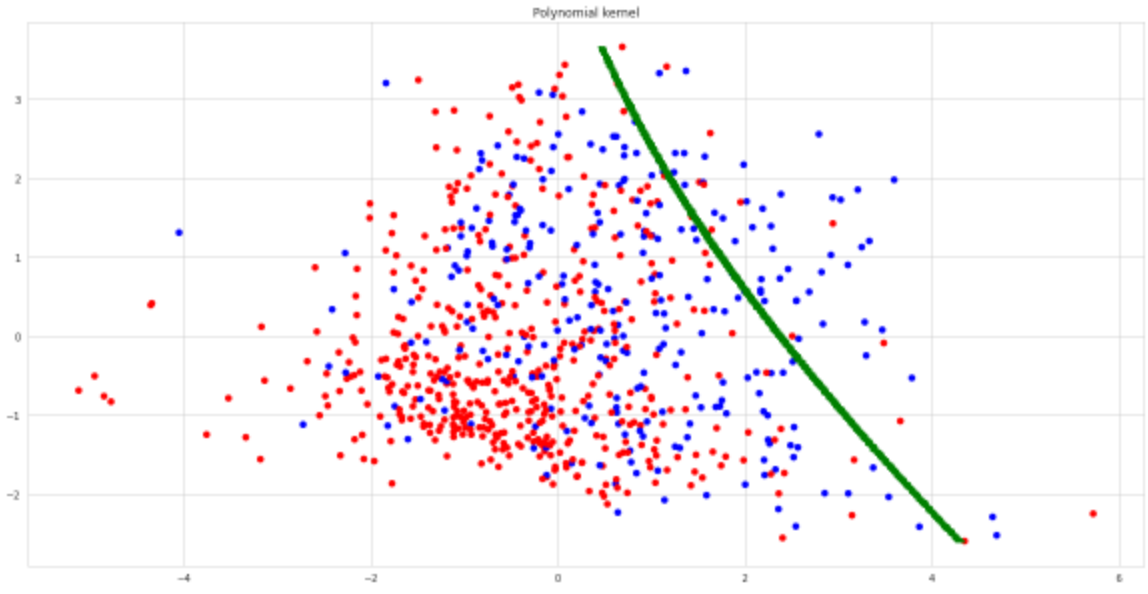
We take the first two principal components ,
Applying SVM on the transformed data :

```
clf_gauss_pca= svm.SVC(kernel='rbf', C= 2.23)
```

Similarly SVM can be applied for linear and polynomial kernels.


Gaussian kernel
3
2
1
0
−1
−2
−4
−2
0
2
4
6


Linear kernel
3
2
1
0
−1
−2
−4
−2
0
2
4
6


Polynomial kernel
3
2
1
0
−1
−2
−4
−2
0
2
4
6