Data Preparation & Preprocessing

Step1: Segregate the images into folders based on labels

Function: create_subdirectory

1. The function iterates through each image
2. For each image filters the label from the image name
3. Checks if a directory exists with label name
4. If Yes, then add the image to the directory
5. If No the create a directory and add the image to the directory

Step2: Image Processing

Function: `create_training_data`

1. The function iterates through each image
2. Converts it into greyscale image
3. Resize the image into 32x32 size
4. Append it to the global array *training_data* in the format [image, label]

Step3: Shuffle the *training_data* array so the model can learn better. For this *random.shuffle* library is used.

Step4: Post shuffling the *training_data* array is divided into below 2 lists. (The reason it is done post shuffling is to maintain one to one mapping between the image and the label).

1. *X*: It contains all the 1500 images numerical representation (Each image is 32*32 array filled with values from 0-255)
2. *Y*: It contains the labels for those images.

Step5: Since *X* is a list it is first converted to NumPy array so that can be passed to the models. The NumPy array is then reshaped to add the 3rd dimension to the data so it can be passed to Conv2D model

Code Snippet: `X = np.array(X).reshape(-1, IMG_SIZE, IMG_SIZE, 1)`

Post reshape *X* can be imagined as a table with 1500 (total images) rows. Each row having 32*32 array
Reshape parameters

- **-1**: Indicated that the table can have dynamic number of rows (1500)
- **IMG_Size, IMG_Size**: Indicates the size of data inside each row
- **1**: Even thought we are building Conv2D model the data that is passed to the model must be 3D.

Step6: To standardize the distribution each image is normalized by dividing by 255. This also makes it easier and fast for the models to learn feature.

Step7: Divide the data into *xTrain, xTest, yTrain, yTest* using the *train_test_split* library. *xTrain, yTrain,* is used for training data and *xTest, yTest* is used as testing data.

Step8: Since both *yTrain, yTest* represent categorical labels it is converted to categorical NumPy array using the *to_categorical* library just before passing it to the model.

Feature Extraction using CNN Model

The CNN model is used for feature extraction. Below are the steps performed for extracting the features from CNN.

Step1: CNN Model Definition

The model is defined as below. The activation function here is assigned to ReLu.

```
def getModel():
    cnnmodel = Sequential()

    cnnmodel.add(Conv2D(50, kernel_size = (5, 5), activation=activationFunction,
                input_shape=(IMG_SIZE, IMG_SIZE, 1)))
    cnnmodel.add(Dropout(0.1))
    cnnmodel.add(MaxPooling2D(pool_size=(2,2)))

    cnnmodel.add(Flatten())
    cnnmodel.add(Dropout(0.1))
    cnnmodel.add(Dense(30, activation=activationFunction))
    cnnmodel.add(Dropout(0.1))
    cnnmodel.add(Dense(15, activation=activationFunction))
    cnnmodel.add(Dropout(0.1))
    cnnmodel.add(Dense(3, activation = 'softmax'))
    cnnmodel.compile(optimizer='adam', loss='categorical_crossentropy',
                     metrics=['categorical_accuracy'])
    cnnmodel.summary()
    return cnnmodel

cnnmodel = getModel()
```

Complete Model Summary is shown below

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 28, 28, 50)        1300      
_________________________________________________________________
dropout (Dropout)            (None, 28, 28, 50)        0         
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 50)        0         
_________________________________________________________________
flatten (Flatten)            (None, 9800)              0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 9800)              0         
_________________________________________________________________
dense (Dense)                (None, 30)                294030    
_________________________________________________________________
dropout_2 (Dropout)          (None, 30)                0         
_________________________________________________________________
dense_1 (Dense)              (None, 15)                465       
_________________________________________________________________
dropout_3 (Dropout)          (None, 15)                0         
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 48        
=================================================================
Total params: 295,843
Trainable params: 295,843
Non-trainable params: 0
```

Step2: CNN training

The CNN model is trained on *xTrain and new_yTrain* (categorical *yTrain*) with below parameters

- Epoch: 20
- Batch Size: 32
- Validation Split: 0.2

Early stopping and Model checkpoint are used during training to avoid overfitting and save the model at intermediate steps. There is no overfitting during model training as there is not major difference between training and validation accuracy and loss.
Below is the output of last few epochs.
(The CNN is trained so weights are assigned to it layers)

```
Epoch 18/20
30/30 [==============================] - ETA: 0s - loss: 0.0298 - 
categorical_accuracy: 0.9875
Epoch 00018: loss did not improve from 0.00461
30/30 [==============================] - 1s 44ms/step - loss: 0.0298 - 
categorical_accuracy: 0.9875 - val_loss: 0.0049 - val_categorical_accuracy: 
1.0000
```

```
Epoch 19/20
29/30 [============================>.] - ETA: 0s - loss: 0.0188 -
categorical_accuracy: 0.9968
Epoch 00019: loss did not improve from 0.00461
30/30 [==============================] - 1s 43ms/step - loss: 0.0182 -
categorical_accuracy: 0.9969 - val_loss: 7.0261e-04 -
val_categorical_accuracy: 1.0000
Epoch 20/20
29/30 [============================>.] - ETA: 0s - loss: 0.0058 -
categorical_accuracy: 0.9989
Epoch 00020: loss did not improve from 0.00461
30/30 [==============================] - 1s 43ms/step - loss: 0.0056 -
categorical_accuracy: 0.9990 - val_loss: 1.9835e-04 -
val_categorical_accuracy: 1.0000
Epoch 00020: early stopping
```

Step3: Feature Layer Extraction

For this assignment the Dense layer (dense_1) just above the Dense layer with softmax (dense_2) is used as the feature extraction layer. The feature layer is extracted is using below code. (-3 represents the third last layer)

```
layer_name = cnnmodel.layers[-3].name

feature_layer = Model(inputs=cnnmodel.input,outputs=cnnmodel.get_layer(l
ayer_name).output)
```

Step4: Training and Testing Feature Extraction

Now the goal is to replace the last Fully connected layer of CNN first with Random forest classifier and then with KNN classifier. To achieve this, we extract features from the above-mentioned CNN layer and pass it to Random Forest and KNN classifiers.
Features are extracted separately for the training and testing phase of Random Forest and KNN.

- Training features are extracted by running predict function on extracted feature layer i.e dense_1 with *xTrain* data.

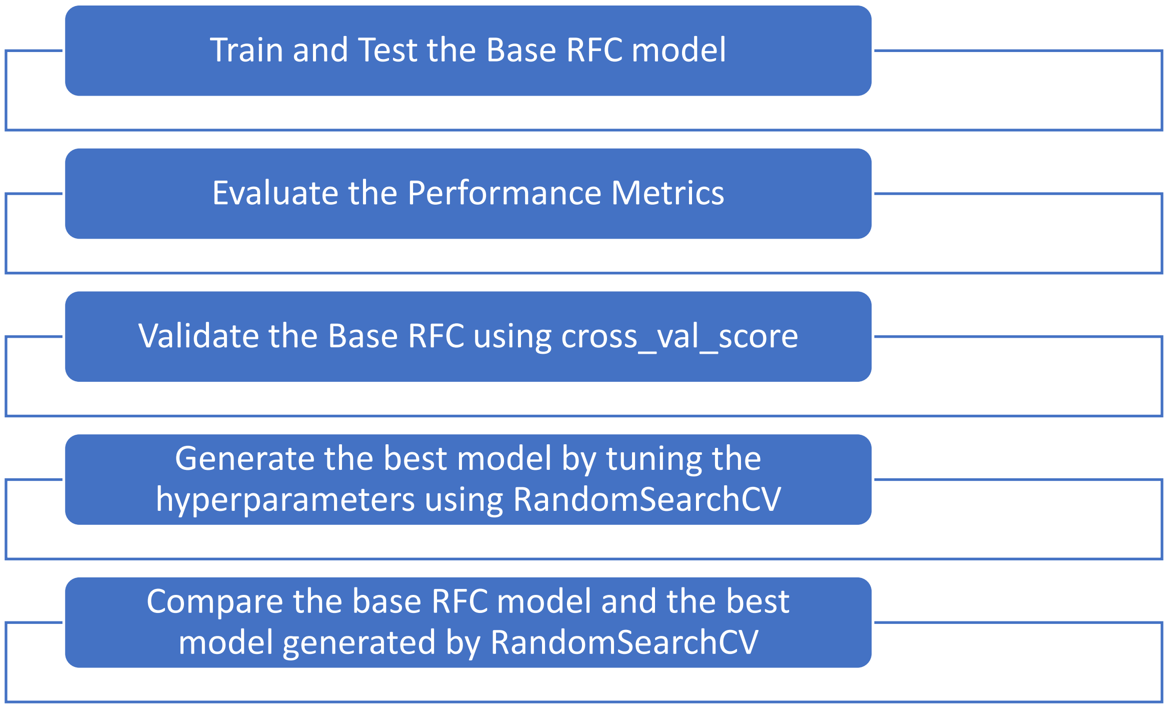
  ```
  train_data_features=feature_layer.predict(xTrain,verbose=1)
  ```

- Testing features are extracted by running predict function on the extracted feature layer i.e dense_1 with *xTest* data.

  ```
  test_data_features=feature_layer.predict(xTest,verbose=1)
  ```

Random Forest Classifier

Below is the summarization of overall flow followed while implementing the Random Forest Classifier (RFC).


Train and Test the Base RFC model
Evaluate the Performance Metrics
Validate the Base RFC using cross_val_score
Generate the best model by tuning the hyperparameters using RandomSearchCV
Compare the base RFC model and the best model generated by RandomSearchCV

<u>**Step1:**</u> Base RFC Training

The base Random Forest Classifier (RFC) is defined using the below hyperparameters and are trained on the Training features extracted by dense_1. (random_state is also defined to replicate the results)

- n_estimators: 20
- max_depth: 100
- max_features: auto

<u>Step2:</u> Base RFC Testing

The base RFC is Tested on the Testing features extracted by dense_1. Below are the performance metrics for the base RFC model

```
base_rf_predict = np.argmax(base_rf_predictions, axis=1)
base_rf_actual = np.argmax(new_yTest, axis=1)
showResults("Base RF",base_rf_actual, base_rf_predict)
```

```
Accuracy  : 1.0
Precision : 1.0
f1Score : 1.0
Confusion Matrix
[[ 99   0   0]
 [  0  93   0]
 [  0   0 108]]
```

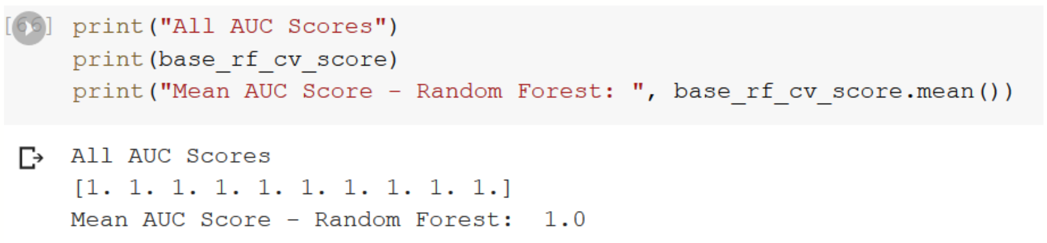
The above result indicates

- Accuracy 100% means 100% of the images were correctly identified
- F1Score (which is calculated using precision and recall) 100% means none of the images were falsely labeled.

The model gives an accuracy of 100%. However, this accuracy is obtained by testing the base RFC model only on a single test set. The model performance can vary when the test set is changed. Thus, using cross_val _score library we perform cross validation on the base RFC and get better idea of the performance of the model since with cross validation the model is tested against k groups of unseen test data.

<u>Step3:</u> Base RFC Validation

10 Fold Statifiedkfold Cross validation is performed using cross_val _score library on the base RFC model and the roc_auc metirc is measured on each fold.
Since for all the 10 folds the auc score is equal to 1 we can say the model works good with unseen data and is not over fitting/ underfitting



```
print("All AUC Scores")
print(base_rf_cv_score)
print("Mean AUC Score - Random Forest: ", base_rf_cv_score.mean())
```

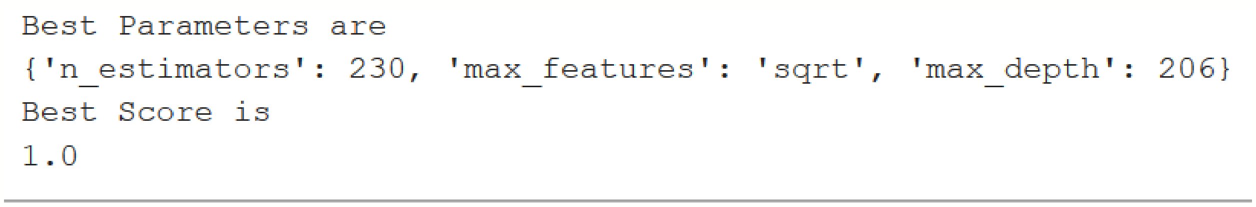
```
All AUC Scores
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
Mean AUC Score - Random Forest:  1.0
```

<u>Step4:</u> Hyperparameter Tuning using RandomsearchCV

Below hyperparameters are tuned using RandomSearchCV library

- n_estimators: In range if 10 to 1000
- max_depth: In range if 20 to 300
- max_features: [auto, sqrt]

The RandomsearchCV gives the below best_params and best_scores. The accuracy is similar to that of base model.



```
Best Parameters are
{'n_estimators': 230, 'max_features': 'sqrt', 'max_depth': 206}
Best Score is
1.0
```

The best model from RandomsearchCV is then extracted and prediction is performed using the Testing features extracted by dense_1

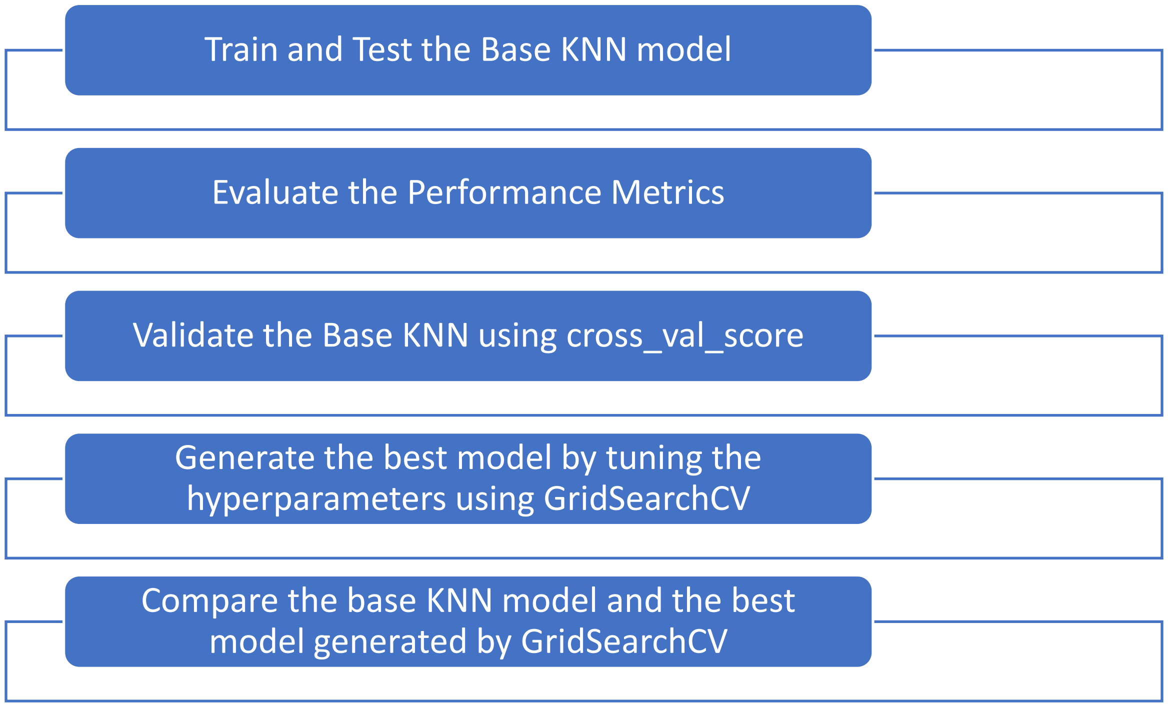
<u>Step5:</u> Comparison of Base RFC and Best RFC model.

```
Base Random Forest Model

Accuracy  : 1.0
Precision : 1.0
f1Score : 1.0
Confusion Matrix
[[ 99   0   0]
 [  0  93   0]
 [  0   0 108]]
All AUC Scores
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
Mean AUC Score - Random Forest:  1.0

RandomSearchCV Model

Accuracy  : 1.0
Precision : 1.0
f1Score : 1.0
Confusion Matrix
[[ 99   0   0]
 [  0  93   0]
 [  0   0 108]]
All AUC Scores
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Both base RFC model and Best RFC share the same performance. One possible reason can be that the dataset is small and too easy to learn.

KNN Classifier

Below is the summarization of overall flow followed while implementing the KNN Classifier .


Train and Test the Base KNN model
Evaluate the Performance Metrics
Validate the Base KNN using cross_val_score
Generate the best model by tuning the hyperparameters using GridSearchCV
Compare the base KNN model and the best model generated by GridSearchCV

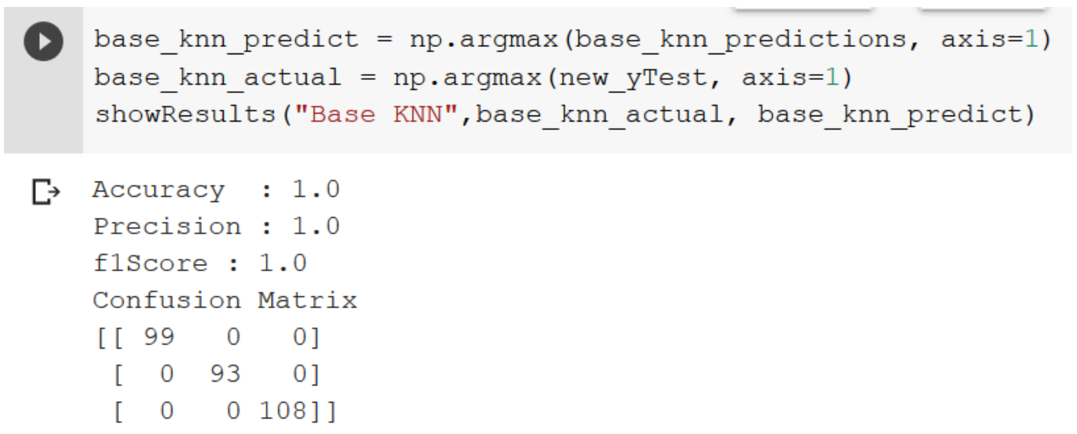
Step1: Base KNN Training

The base KNN is defined using the below hyperparameters and are trained on the Training features extracted by dense_1.

- n_neighbors=3

Step2: Base KNN Testing

The base KNN is Tested on the Testing features extracted by dense_1. Below are the performance metrics for the base KNN model.



```
base_knn_predict = np.argmax(base_knn_predictions, axis=1)
base_knn_actual = np.argmax(new_yTest, axis=1)
showResults("Base KNN",base_knn_actual, base_knn_predict)
```

```
Accuracy  : 1.0
Precision : 1.0
f1Score : 1.0
Confusion Matrix
[[ 99   0   0]
 [  0  93   0]
 [  0   0 108]]
```

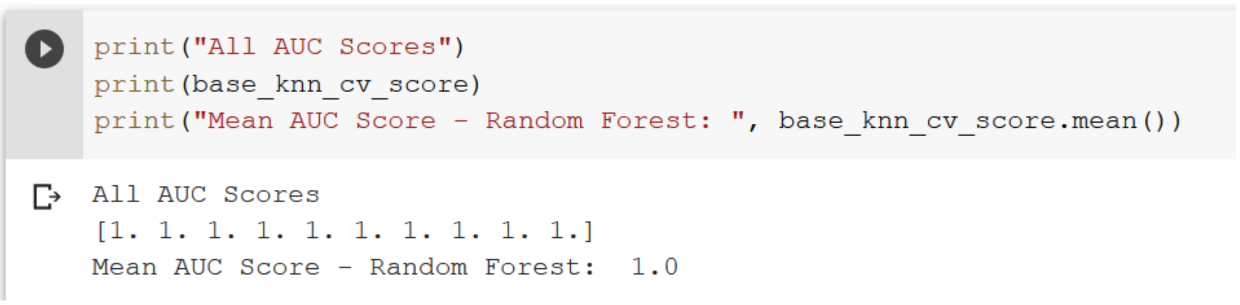
Similar to RFC the above KNN result indicates

- Accuracy 100% means 100% of the images were correctly identified
- F1Score (which is calculated using precision and recall) 100% means none of the images were falsely labeled.

The model gives an accuracy of 100%. However, this accuracy is obtained by testing the base KNN model only on a single test set. The model performance can vary when the test set is changed. Thus, using cross_val _score library we perform cross validation on the base KNN and get better idea of the performance of the model since with cross validation the model is tested against k groups of unseen test data.

Step3: Base KNN Validation

10 Fold Statifiedkfold Cross validation is performed using cross_val _score library on the base KNN model and the roc_auc metirc is measured on each fold.
Since for all the 10 folds the auc score is equal to 1 we can say the model works good with unseen data and is not over fitting/ underfitting.



```
print("All AUC Scores")
print(base_knn_cv_score)
print("Mean AUC Score - Random Forest: ", base_knn_cv_score.mean())
```

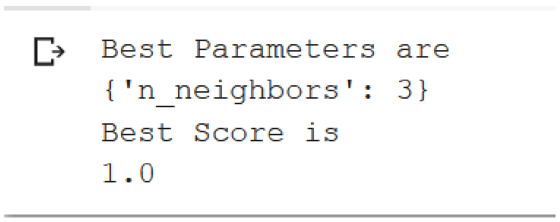
```
All AUC Scores
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
Mean AUC Score - Random Forest:  1.0
```

__Step4:__ Hyperparameter Tuning using GridsearchCV

Below hyperparameters are tuned using GridSearchCV library

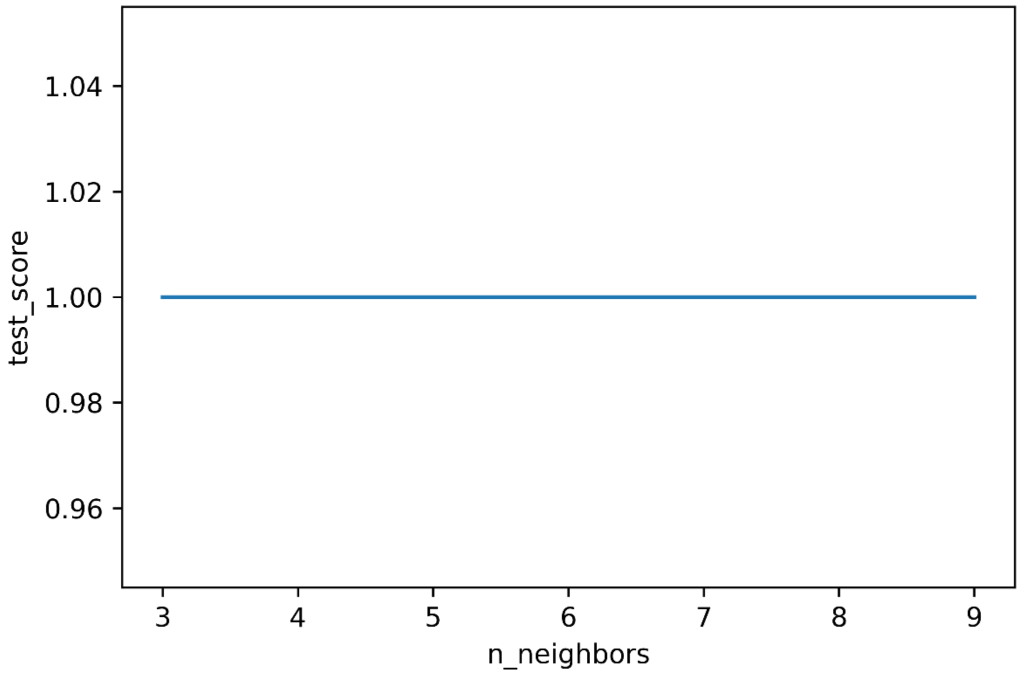
- n_neighbors: [3,5,7,9]

The GridsearchCV gives the below best_params and best_scores. The accuracy is similar to that of base model.



```
Best Parameters are
{'n_neighbors': 3}
Best Score is
1.0
```

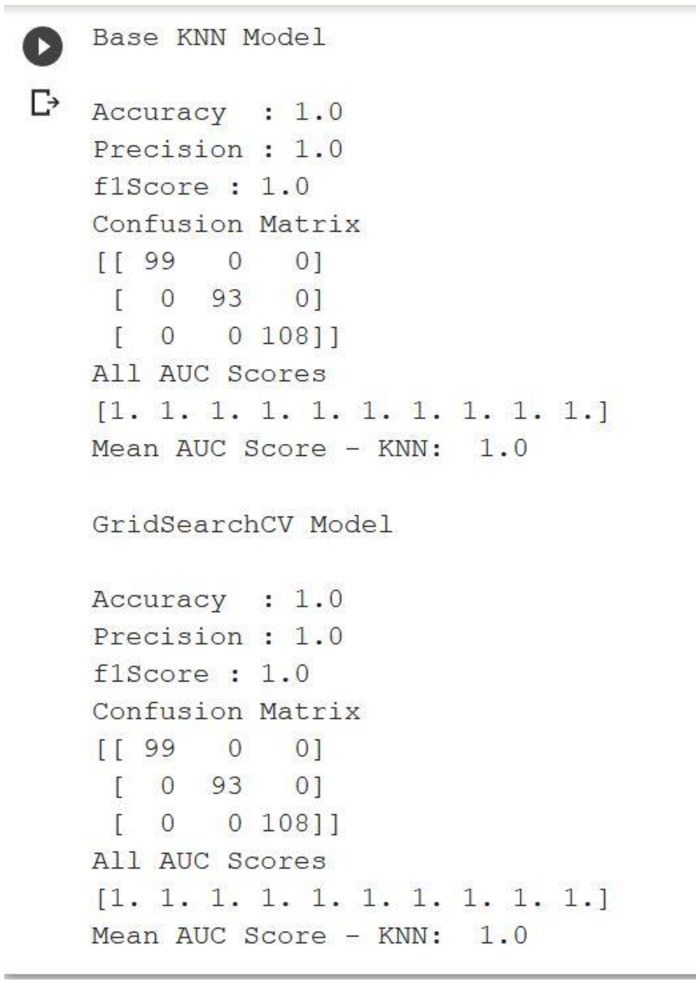
Below graph represents the mean_test_score for different value of k. As we get a straight line the model score remains same for different values of k.



Now the best model from GridsearchCV is extracted and prediction is performed using the Testing features extracted by dense_1.

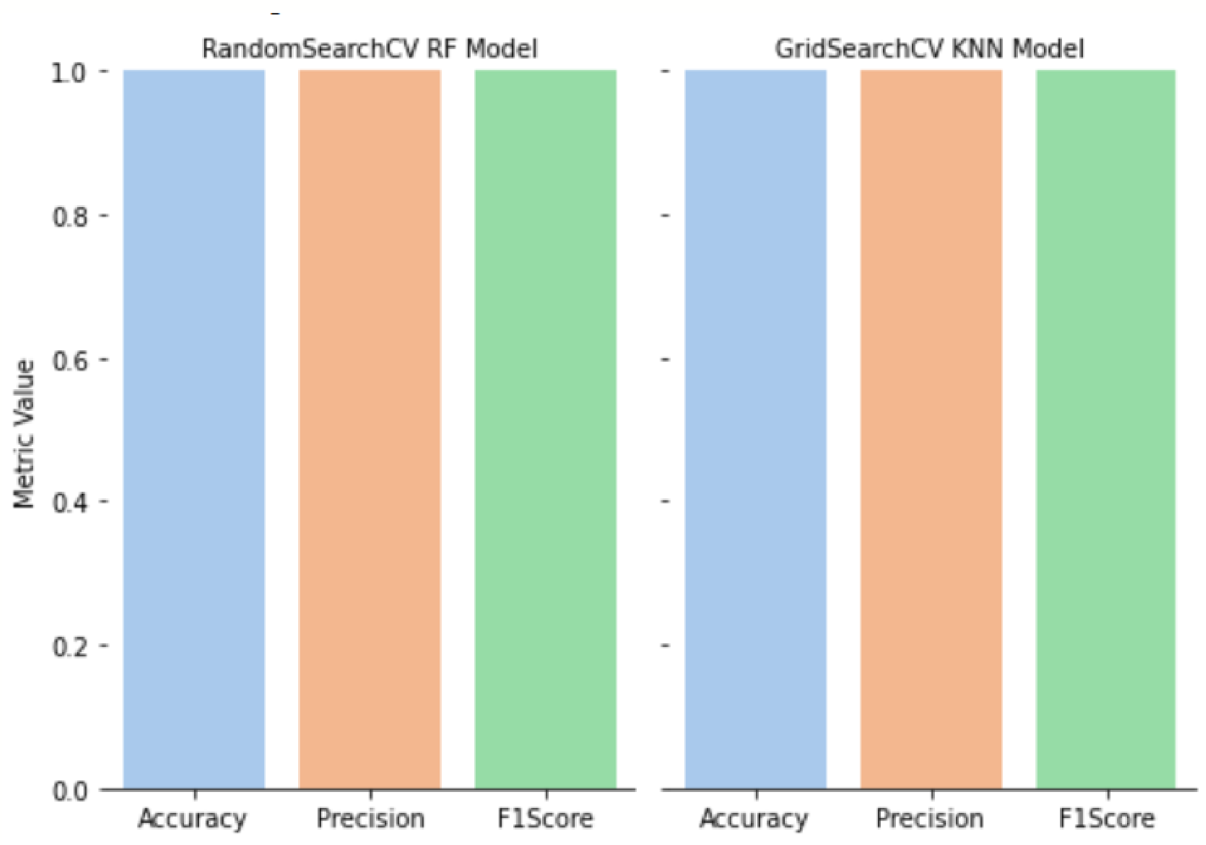
Step5: Comparison of Base KNN and Best KNN model.

Both base KNN model and Best KNN share the same performance



```
Base KNN Model

Accuracy  : 1.0
Precision : 1.0
f1Score : 1.0
Confusion Matrix
[[ 99   0   0]
 [  0  93   0]
 [  0   0 108]]
All AUC Scores
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
Mean AUC Score - KNN:  1.0

GridSearchCV Model

Accuracy  : 1.0
Precision : 1.0
f1Score : 1.0
Confusion Matrix
[[ 99   0   0]
 [  0  93   0]
 [  0   0 108]]
All AUC Scores
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
Mean AUC Score - KNN:  1.0
```

Performance Metrics plot of RandomSearchCV RFC and GridSearchKNN model.


RandomSearchCV RF Model
GridSearchCV KNN Model
Metric Value
Accuracy
Precision
F1Score
Accuracy
Precision
F1Score