Machine Learning Project Documentation

Team Number:

68

	Team Member ID	Team member name
1	20210243	تقى محمد محمد ابوالفتوح توفيق
2	20210261	جومانا محمد السيد طلبه
3	20210565	عبدالواحد رجب عبدالواحد حماد
4	20210531	عبدالرحمن مصطفى حامد محمود
5	20210166	الاء حسن عبدالرحمن عبدالرحمن
6	20210172	الاء ممدوح احمد عثمان

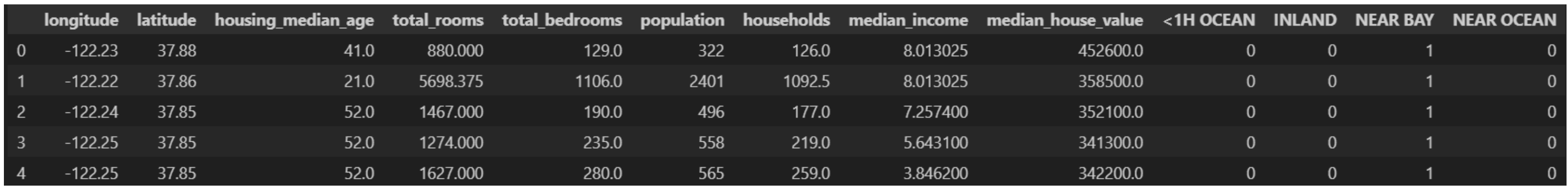
Housing California Price

Link in Kaggle:

California Housing Prices (kaggle.com)

1 Data Summary

Here we first summarize the California Housing dataset using visualization and some basic statistics. As showed in Figure 1California Housing dataset contains 20640 rows and each one of them stores information about a specific block. It contains 13 columns with 12 features and one target variable-median house value.



	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	<1H OCEAN	INLAND	NEAR BAY	NEAR OCEAN
0	-122.23	37.88	41.0	880.000	129.0	322	126.0	8.013025	452600.0	0	0	1	0
1	-122.22	37.86	21.0	5698.375	1106.0	2401	1092.5	8.013025	358500.0	0	0	1	0
2	-122.24	37.85	52.0	1467.000	190.0	496	177.0	7.257400	352100.0	0	0	1	0
3	-122.25	37.85	52.0	1274.000	235.0	558	219.0	5.643100	341300.0	0	0	1	0
4	-122.25	37.85	52.0	1627.000	280.0	565	259.0	3.846200	342200.0	0	0	1	0

About Dataset:

1.**longitude**: A measure of how far west a house is; a higher value is farther west

2. **latitude**: A measure of how far north a house is; a higher value is farther north

3. **housingMedianAge**: Median age of a house within a block; a lower number is a newer building

4. **totalRooms**: Total number of rooms within a block

5. **totalBedrooms**: Total number of bedrooms within a block

6. **population**: Total number of people residing within a block

7. **households**: Total number of households, a group of people residing within a home unit, for a block

8. **medianIncome**: Median income for households within a block of houses (measured in tens of thousands of US Dollars)

9. **medianHouseValue**: Median house value for households within a block (measured in US Dollars)

10. **oceanProximity**: Location of the house w.r.t ocean/sea
Acknowledgements

Inspiration

See my kernel on machine learning basics in R using this dataset, or venture over to the following link for a python based introductory tutorial: https://github.com/ageron/handson-ml/tree/master/datasets/housing

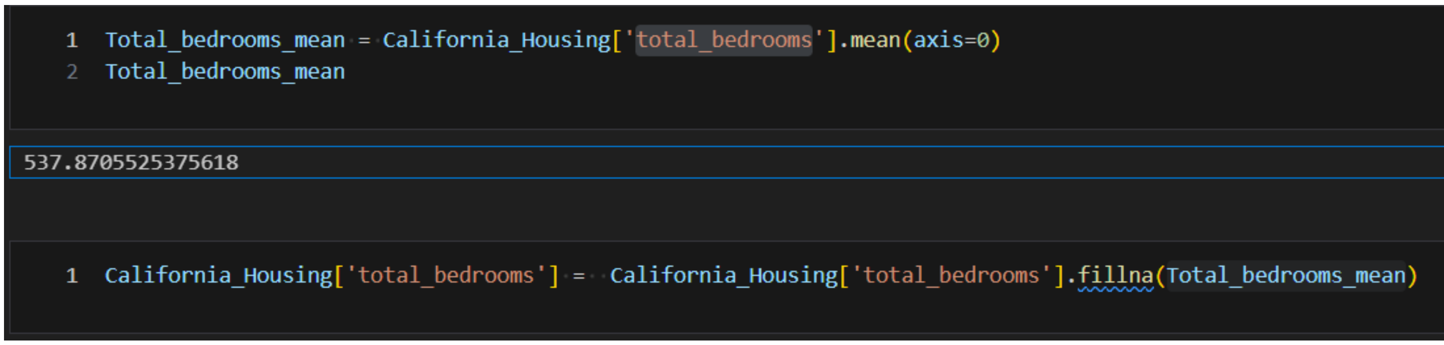
Preprocessing steps:

1- Handling missing values:

Table 1. Number of NAN values of all Variables

Variable	Description
longitude	0
latitude	0
housing median age	0
total rooms	0
total bedrooms	207
population	0
households	0
median income	0
ocean proximity	0
median house value	0

- In feature total_bedrooms there are missing values To handle this missing values we fill this missing values with mean for feature total_bedrooms the code below explain how we do this:



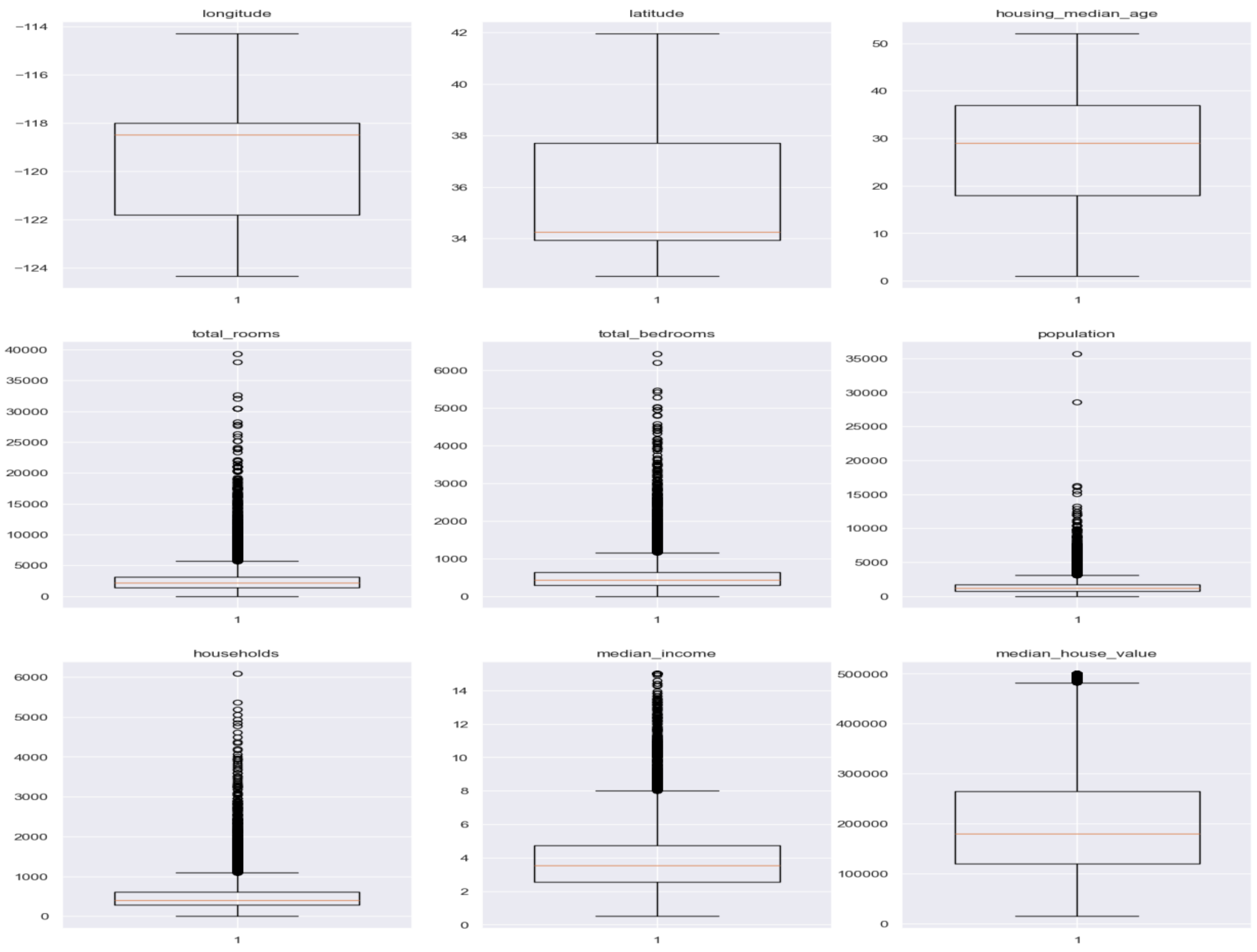
```
Total_bedrooms_mean = California_Housing['total_bedrooms'].mean(axis=0)
Total_bedrooms_mean
```

```
537.8705525375618
```

```
California_Housing['total_bedrooms'] = California_Housing['total_bedrooms'].fillna(Total_bedrooms_mean)
```

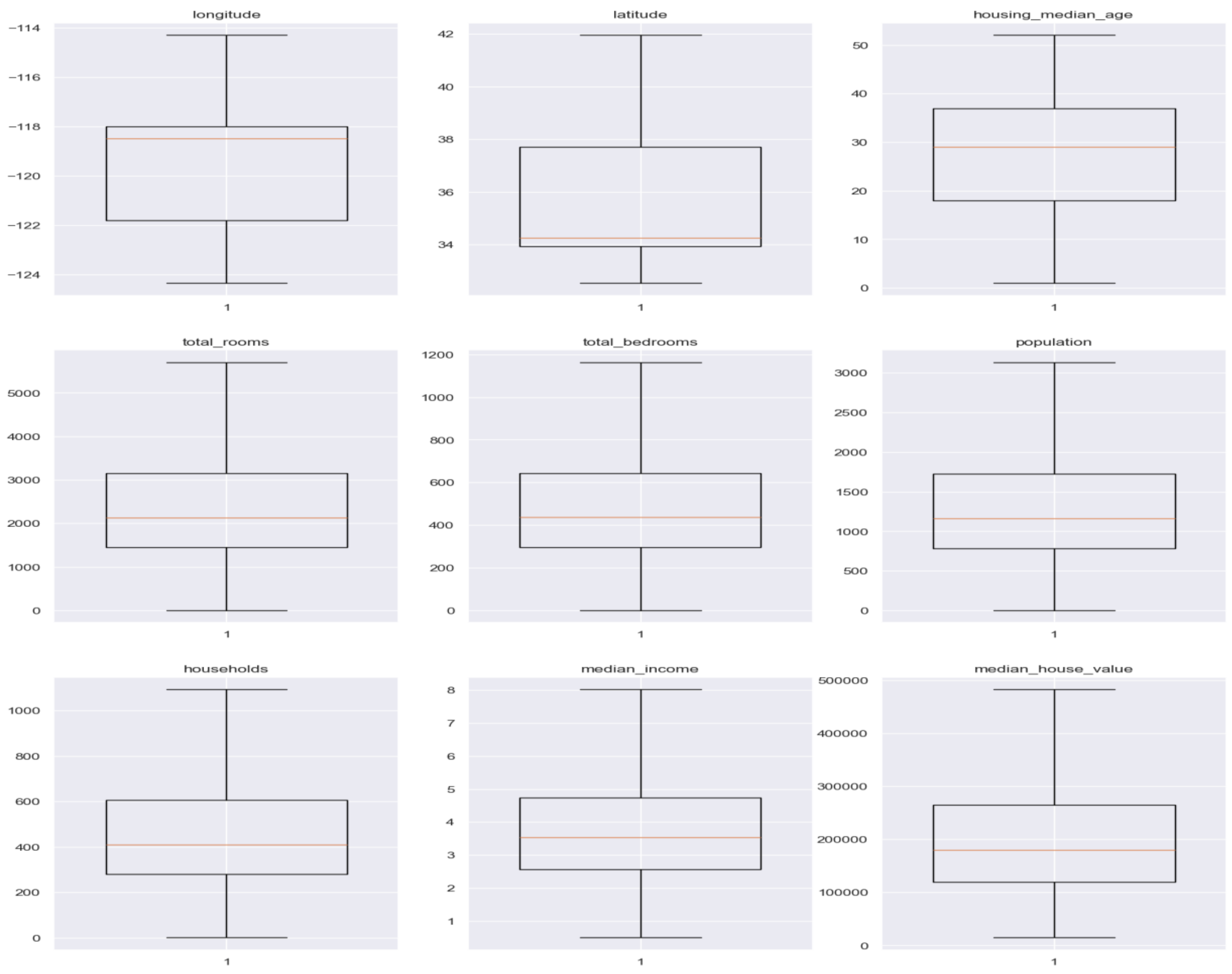
2- handling Outliers with IQR method

- detected if their outliers using boxenplots so there the shape of some of columns before deleting



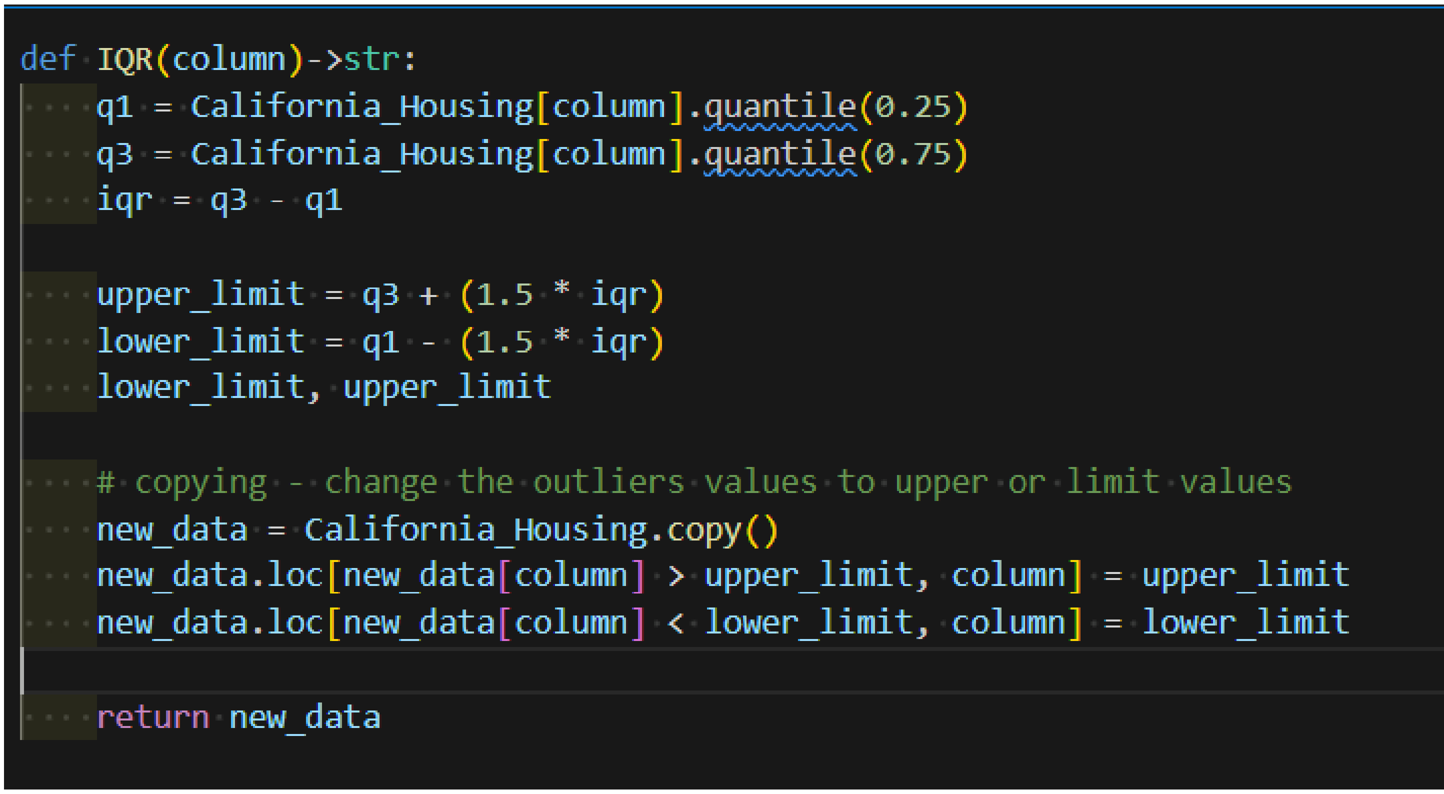
outliers:

- And this is the shape after deleting.



- Quantile based flooring and capping: In this technique, the outlier is capped at a certain value above the 90th percentile value or floored at a factor below the 10th percentile value.

- We use the Quantile based flooring and capping to handle the outliers the code below explain how we



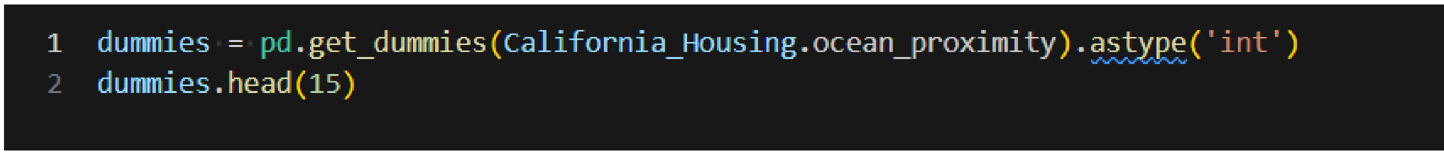
```python
def IQR(column)->str:
    q1 = California_Housing[column].quantile(0.25)
    q3 = California_Housing[column].quantile(0.75)
    iqr = q3 - q1

    upper_limit = q3 + (1.5 * iqr)
    lower_limit = q1 - (1.5 * iqr)
    lower_limit, upper_limit

    # copying - change the outliers values to upper or limit values
    new_data = California_Housing.copy()
    new_data.loc[new_data[column] > upper_limit, column] = upper_limit
    new_data.loc[new_data[column] < lower_limit, column] = lower_limit

    return new_data
```

do that:

3- Handling Categoriacal values (using encoding)

- In machine learning, categorical values refer to variables that can take on a limited, and usually fixed, number of possible values. These values typically represent categories or labels, such as "red," "blue," "green" for a color variable. Handling categorical values in machine learning involves techniques such as one-hot encoding, label encoding, or using techniques like decision trees that can naturally handle categorical data

- The code below explain how we handling categorical values



```python
dummies = pd.get_dummies(California_Housing.ocean_proximity).astype('int')
dummies.head(15)
```

4- Feature Scalling

- In machine learning, feature scaling is the process of normalizing or standardizing the range of independent variables or features of the data. It is important because many machine learning algorithms perform better when the input numerical variables are scaled to a standard range. Common techniques for feature scaling include min-max scaling, standardization (Z-score normalization), and robust scaling. These techniques help to ensure that all features have a similar scale and do not disproportionately influence the learning algorithm.

- The code below explain how to make feature scalling

```python
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)
```

5- Spliting Data to Train,Test

- Splitting data into training and testing sets is a fundamental practice in machine learning. This involves dividing the available dataset into two separate sets: the training set, which is used to train the model, and the testing set, which is used to evaluate the model's performance. Typically, a larger portion of the data is allocated to the training set (e.g., 75%), while the remaining portion is used for testing. This process allows for assessing how well the trained model generalizes to new, unseen data.

- The code below explain how we make splitting data to train and test data.

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 0)
```

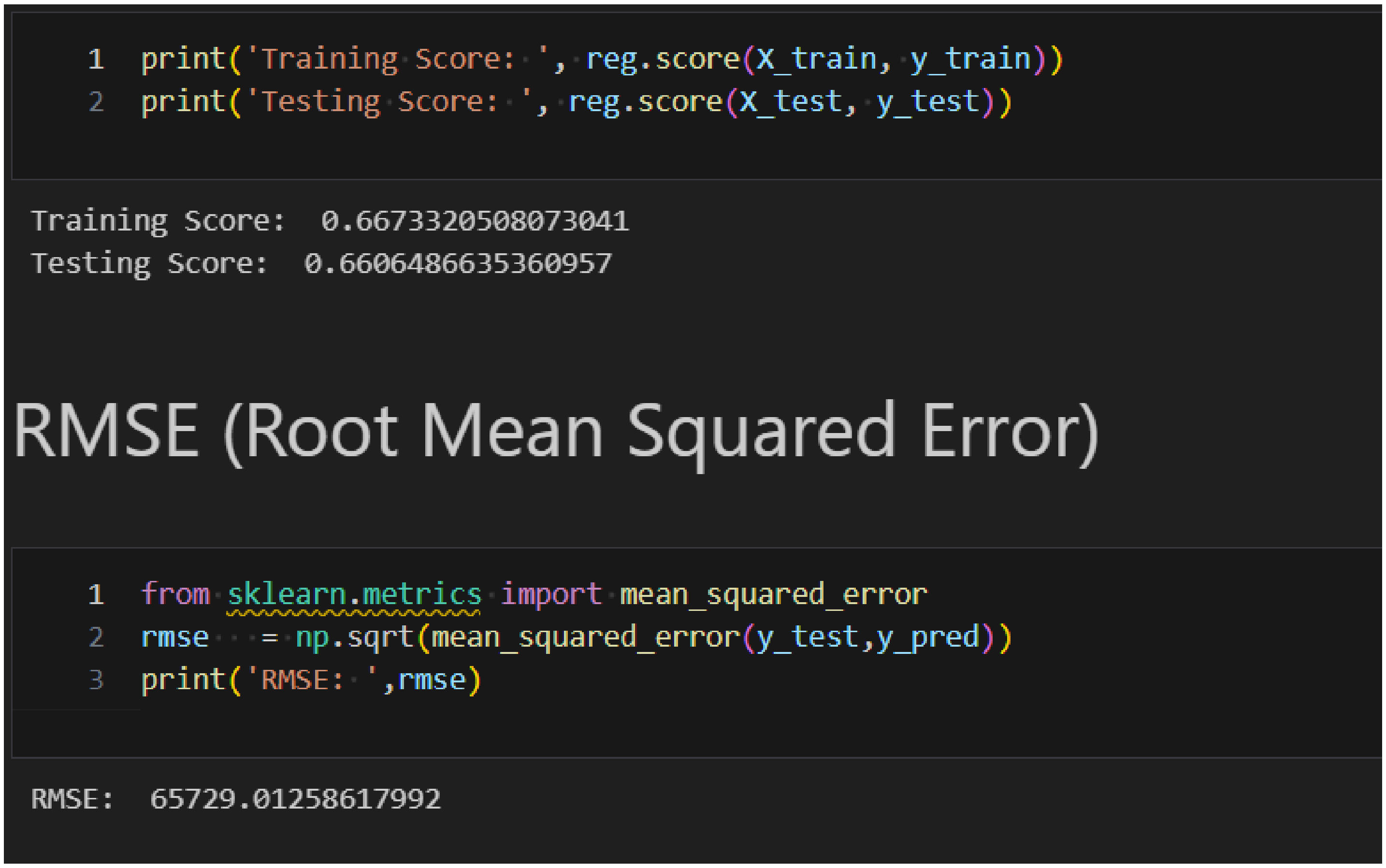
The data is ready to be in the models.

a) Linear Regression

b) Goal : **median_house_value**

c) Code : **used linear regression from sklearn.lineae_model**

d) Result



```
print('Training Score: ', reg.score(X_train, y_train))
print('Testing Score: ', reg.score(X_test, y_test))
```

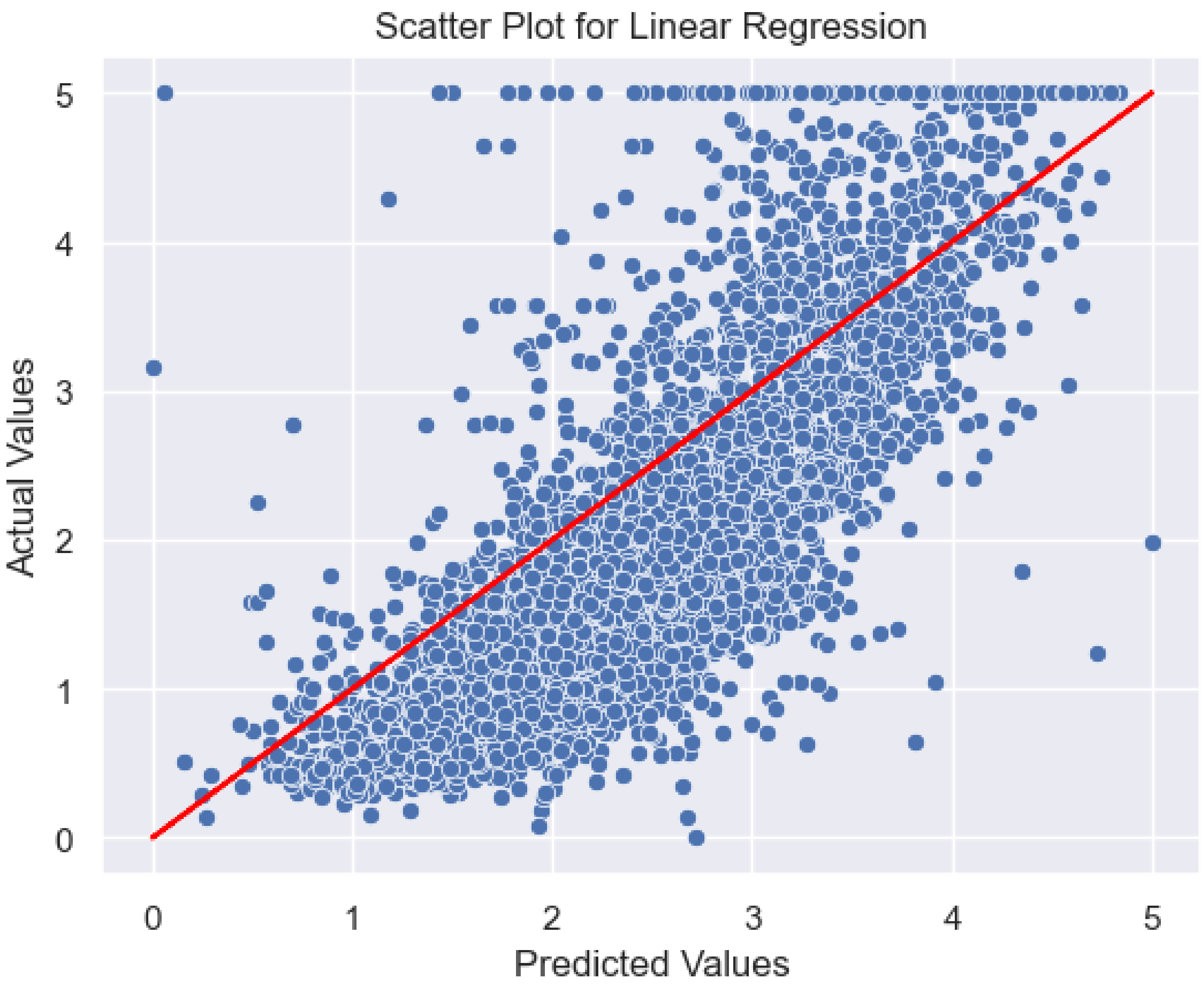
```
Training Score:  0.6673320508073041
Testing Score:  0.6606486635360957
```

RMSE (Root Mean Squared Error)

```
from sklearn.metrics import mean_squared_error
rmse   = np.sqrt(mean_squared_error(y_test,y_pred))
print('RMSE: ',rmse)
```

```
RMSE:  65729.01258617992
```

And this the plotting of the model:

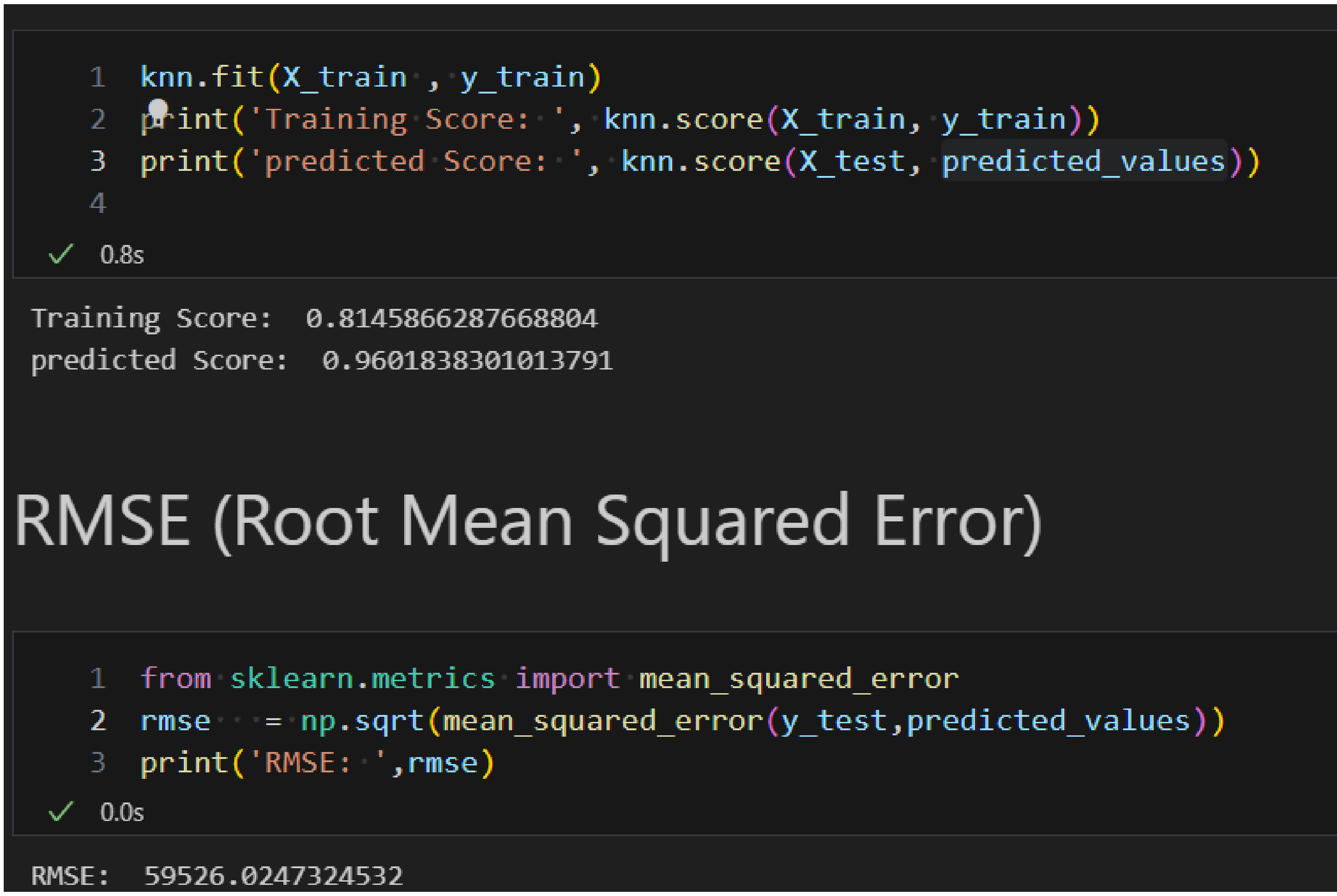


a) k Nearest Neighbors (KNN)

b) Goal : **median_house_value**

c) Code : **used** KNeighborsRegressor from sklearn. neighbors

d) Result



```
knn.fit(X_train , y_train)
print('Training Score: ', knn.score(X_train, y_train))
print('predicted Score: ', knn.score(X_test, predicted_values))

✓ 0.8s

Training Score:  0.8145866287668804
predicted Score:  0.9601838301013791
```

RMSE (Root Mean Squared Error)

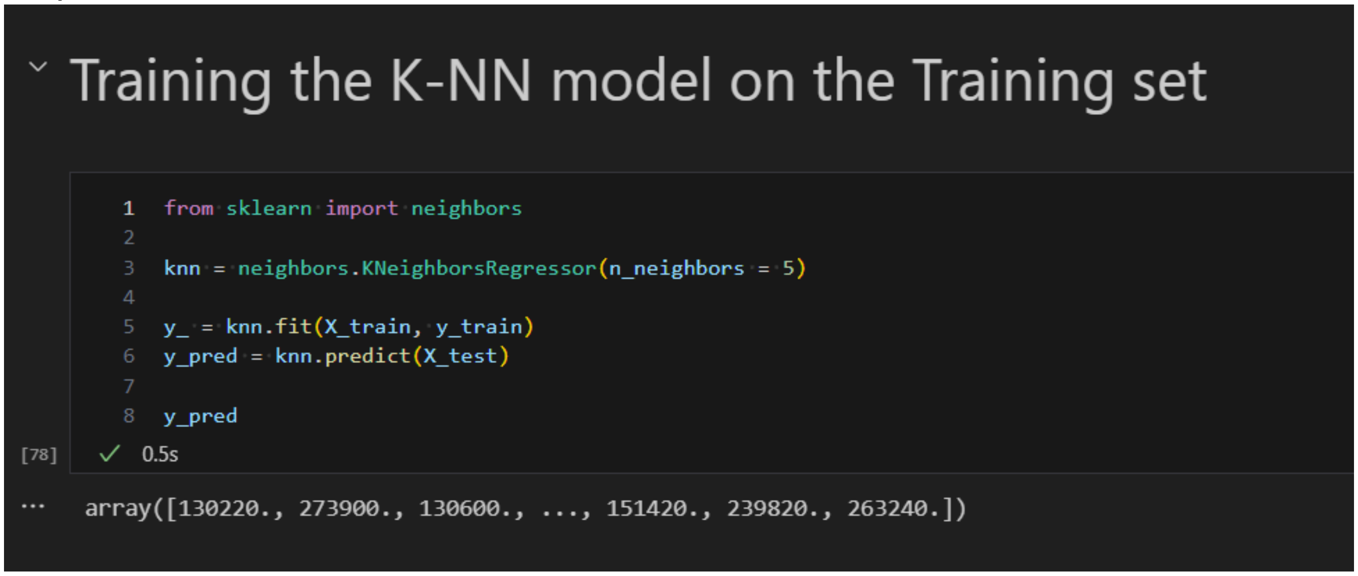
```
from sklearn.metrics import mean_squared_error
rmse   = np.sqrt(mean_squared_error(y_test,predicted_values))
print('RMSE: ',rmse)
✓ 0.0s

RMSE:  59526.0247324532
```

And this the plotting of the
K Nearest Neighbors (KNN)

Purpose : KNN used for regression tasks to predict continuous values based on the average or weighted average of the values of its nearest neighbors. It is particularly useful when the relationship between input features and output is non-linear.

Goal : **median_house_value**

Implementation :



Training the K-NN model on the Training set

```
from sklearn import neighbors

knn = neighbors.KNeighborsRegressor(n_neighbors = 5)

y_ = knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)

y_pred
```

[78] ✓ 0.5s

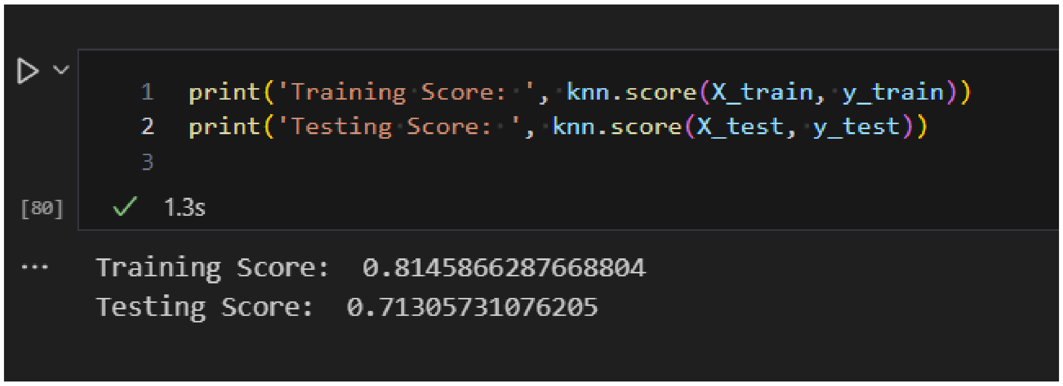
```
array([130220., 273900., 130600., ..., 151420., 239820., 263240.])
```

1.imports the KNeighborsRegressor class from the neighbors module in scikit-learn.

2.set number of neighbours to 5

3. fit the model on training data set

4. predict the model

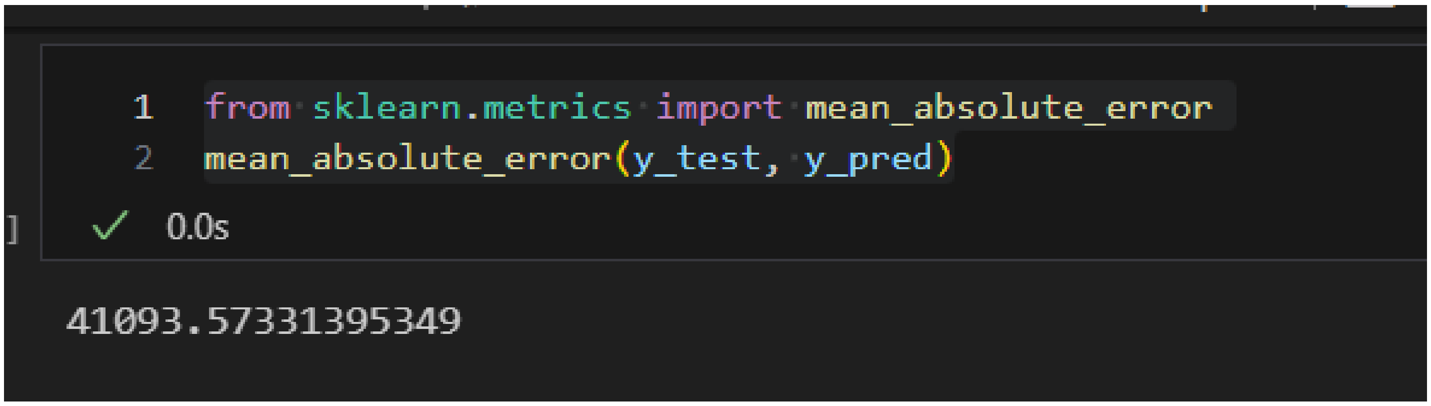


```
print('Training Score: ', knn.score(X_train, y_train))
print('Testing Score: ', knn.score(X_test, y_test))
```

[80] ✓ 1.3s

```
Training Score:  0.8145866287668804
Testing Score:  0.71305731076205
```

The predicted accuracy is 70%



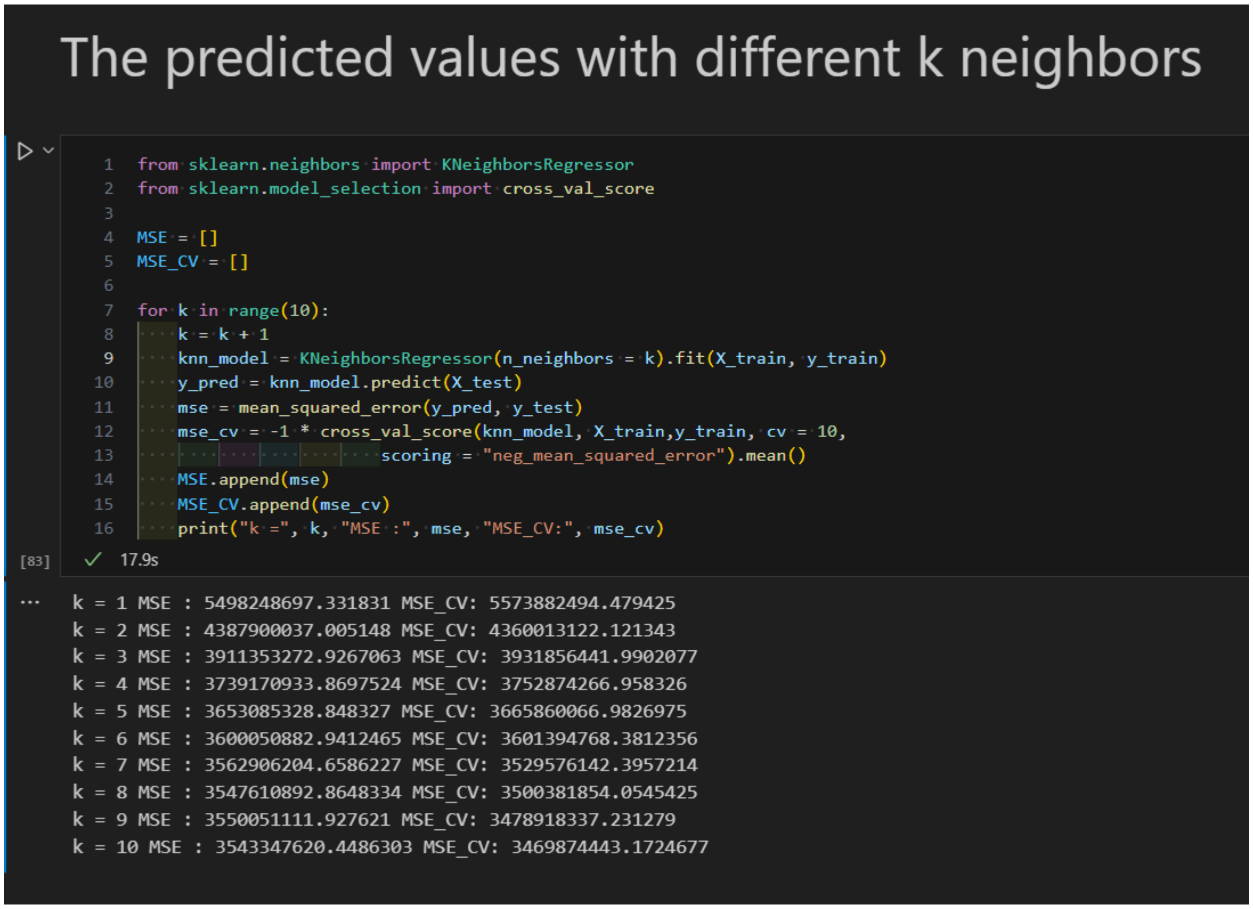
```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, y_pred)
```

✓ 0.0s

```
41093.57331395349
```

This is the result after calculating mean squared error between actual and predicted values (average of error during prediction)

If we want to use different k to set lowest mean squared error can occurs



The predicted values with different k neighbors

```python
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import cross_val_score

MSE = []
MSE_CV = []

for k in range(10):
    k = k + 1
    knn_model = KNeighborsRegressor(n_neighbors = k).fit(X_train, y_train)
    y_pred = knn_model.predict(X_test)
    mse = mean_squared_error(y_pred, y_test)
    mse_cv = -1 * cross_val_score(knn_model, X_train,y_train, cv = 10,
                            scoring = "neg_mean_squared_error").mean()
    MSE.append(mse)
    MSE_CV.append(mse_cv)
    print("k =", k, "MSE :", mse, "MSE_CV:", mse_cv)
```

[83] ✓ 17.9s

```
k = 1 MSE : 5498248697.331831 MSE_CV: 5573882494.479425
k = 2 MSE : 4387900037.005148 MSE_CV: 4360013122.121343
k = 3 MSE : 3911353272.9267063 MSE_CV: 3931856441.9902077
k = 4 MSE : 3739170933.8697524 MSE_CV: 3752874266.958326
k = 5 MSE : 3653085328.848327 MSE_CV: 3665860066.9826975
k = 6 MSE : 3600050882.9412465 MSE_CV: 3601394768.3812356
k = 7 MSE : 3562906204.6586227 MSE_CV: 3529576142.3957214
k = 8 MSE : 3547610892.8648334 MSE_CV: 3500381854.0545425
k = 9 MSE : 3550051111.927621 MSE_CV: 3478918337.231279
k = 10 MSE : 3543347620.4486303 MSE_CV: 3469874443.1724677
```

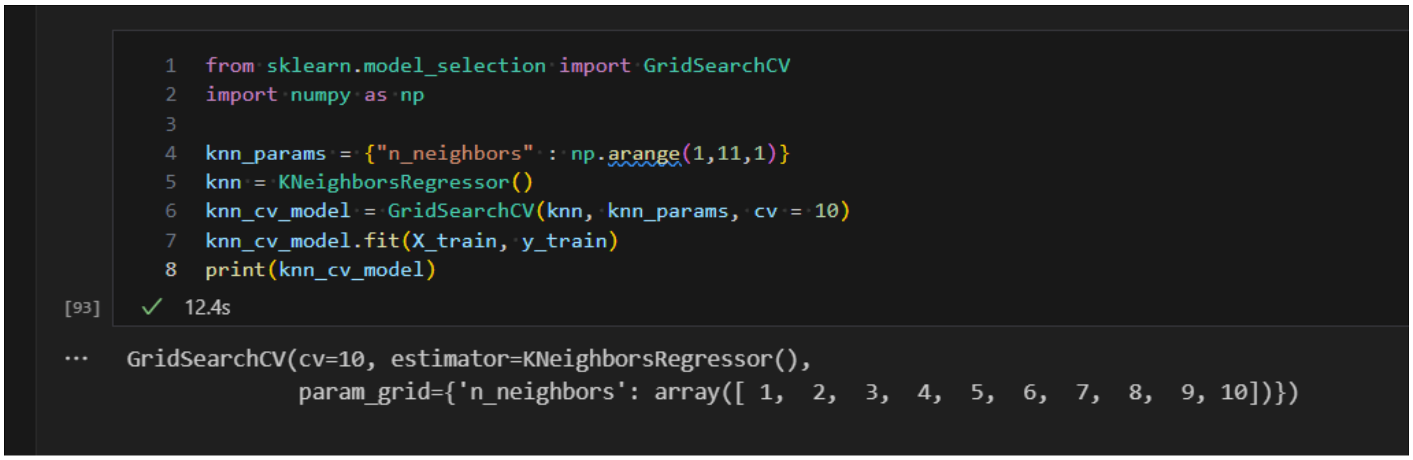
When using mean squared error cross validation it divides the set into k folds
And it gets the mean squared error for each fold for every cv (number of neighbours chosed)
We are using mean squared error (MSE) cross-validation to evaluate the performance of the KNeighborsRegressor

model with different values of k.

MSE measures the average of the squares of the errors or deviations, which gives more weight to larger errors.

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample. It involves splitting the data into multiple subsets, training the model on a subset, and evaluating it on the remaining subset.

By using MSE cross-validation, we can assess how well the model generalizes to new data and choose the optimal value of k for the KNeighborsRegressor model.

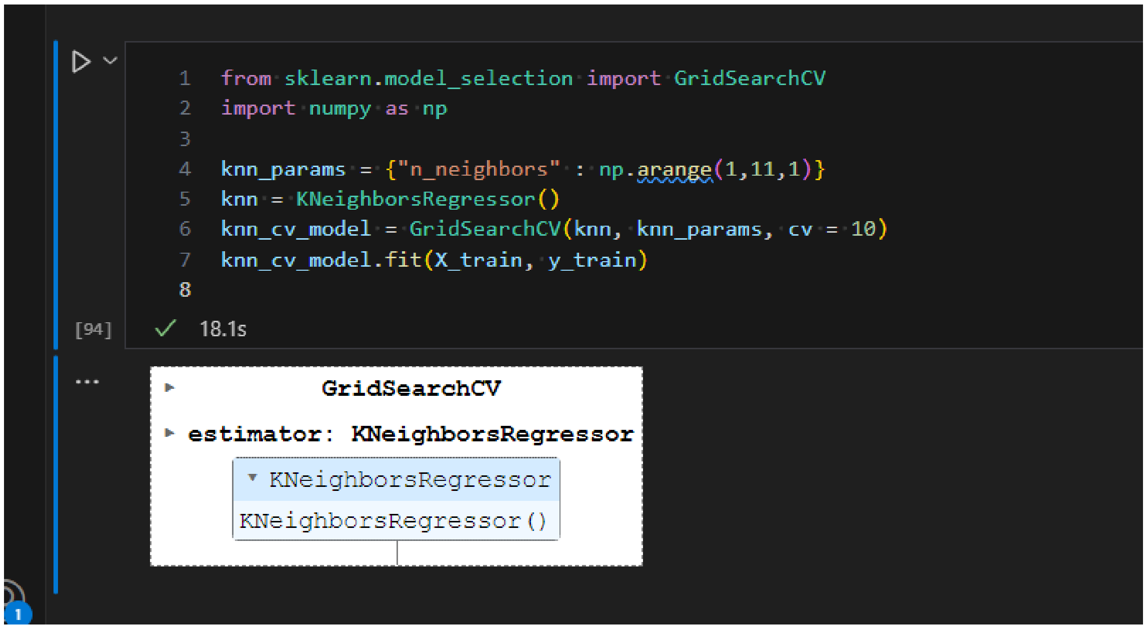


```python
from sklearn.model_selection import GridSearchCV
import numpy as np

knn_params = {"n_neighbors" : np.arange(1,11,1)}
knn = KNeighborsRegressor()
knn_cv_model = GridSearchCV(knn, knn_params, cv = 10)
knn_cv_model.fit(X_train, y_train)
print(knn_cv_model)
```

[93] ✓ 12.4s

```
GridSearchCV(cv=10, estimator=KNeighborsRegressor(),
             param_grid={'n_neighbors': array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])})
```

Using of grid search cv to determine best number of neighbours can use

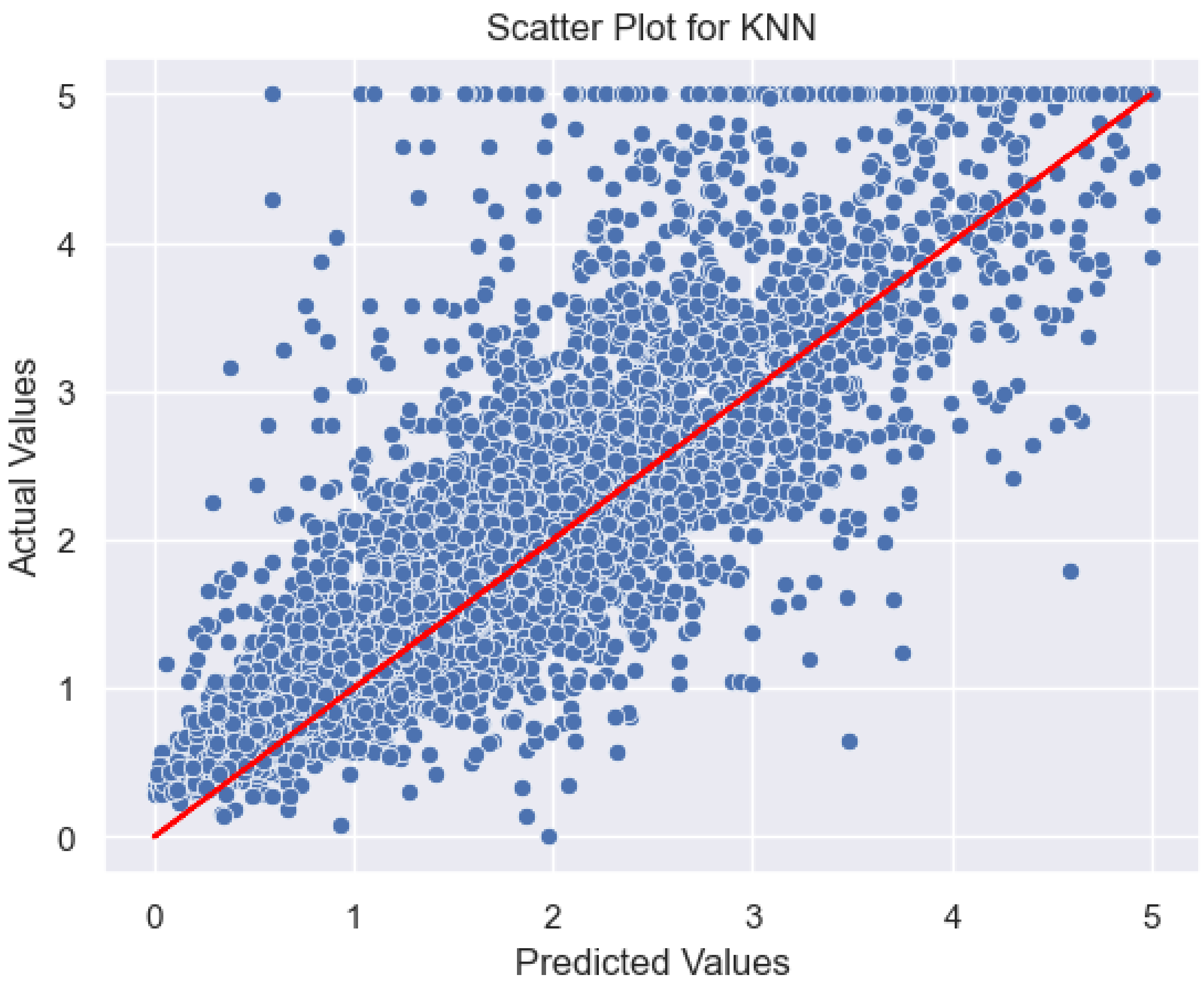


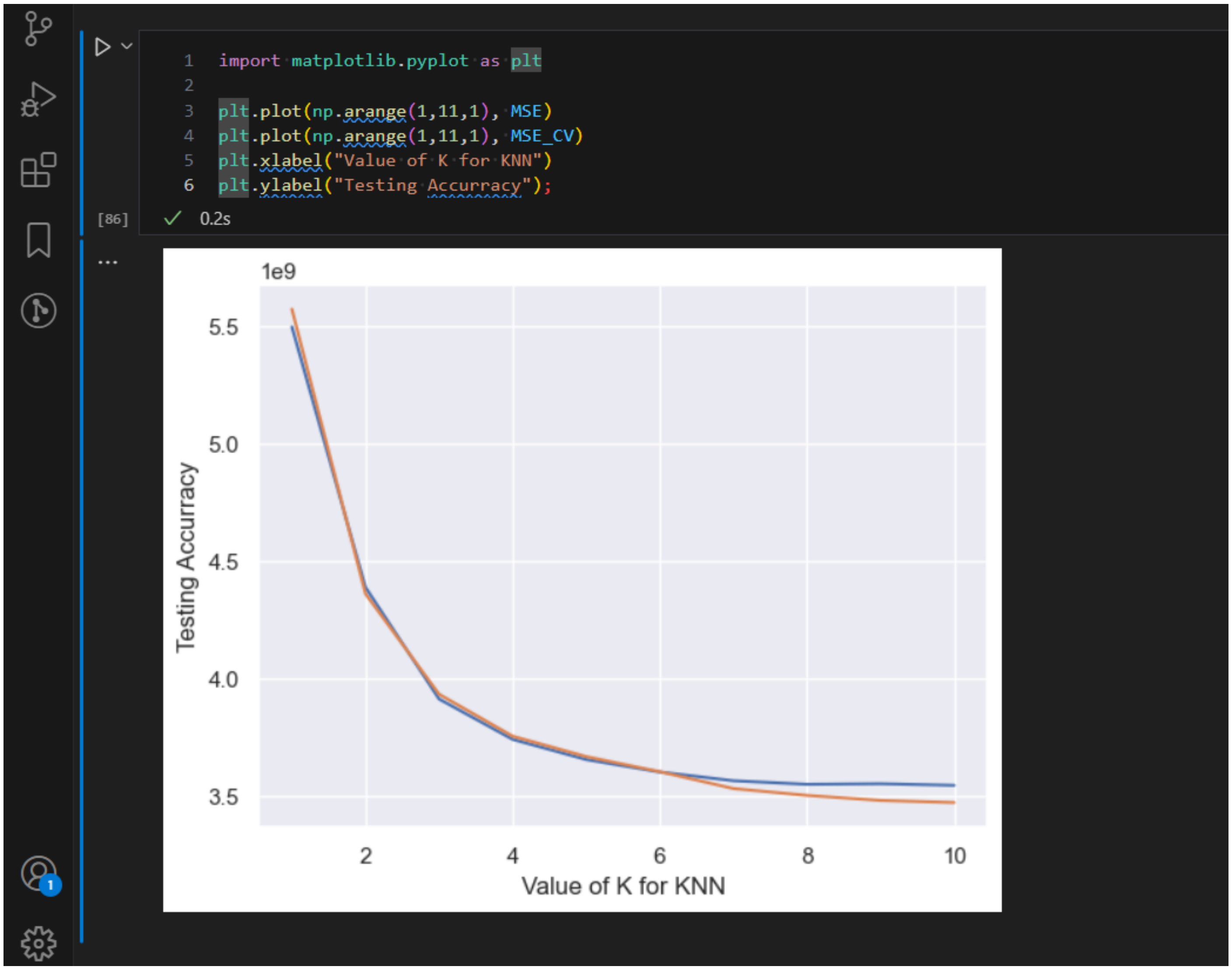
```python
from sklearn.model_selection import GridSearchCV
import numpy as np

knn_params = {"n_neighbors" : np.arange(1,11,1)}
knn = KNeighborsRegressor()
knn_cv_model = GridSearchCV(knn, knn_params, cv = 10)
knn_cv_model.fit(X_train, y_train)

```

[94] ✓ 18.1s

```
GridSearchCV
estimator: KNeighborsRegressor
KNeighborsRegressor
KNeighborsRegressor()
```

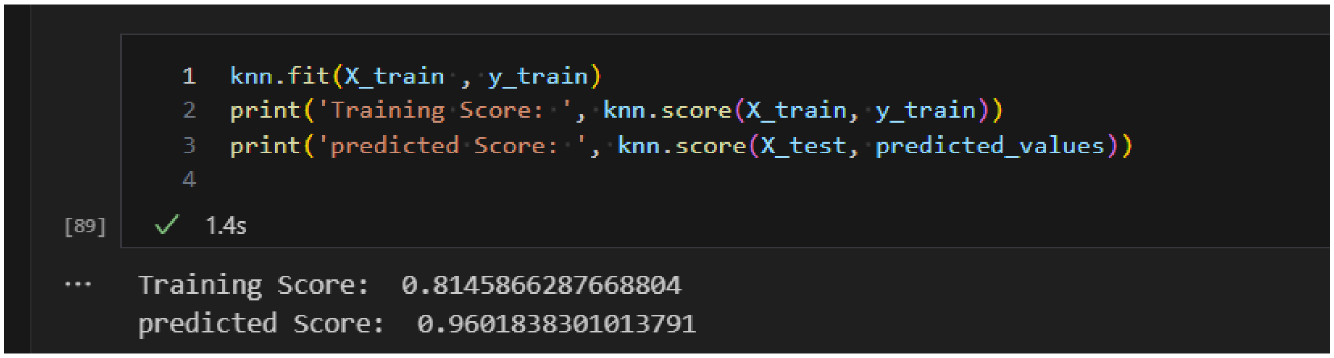
And this the plotting of the model:


Scatter Plot for KNN
Actual Values
Predicted Values
0
1
2
3
4
5



```
import matplotlib.pyplot as plt

plt.plot(np.arange(1,11,1), MSE)
plt.plot(np.arange(1,11,1), MSE_CV)
plt.xlabel("Value of K for KNN")
plt.ylabel("Testing Accurracy");
```

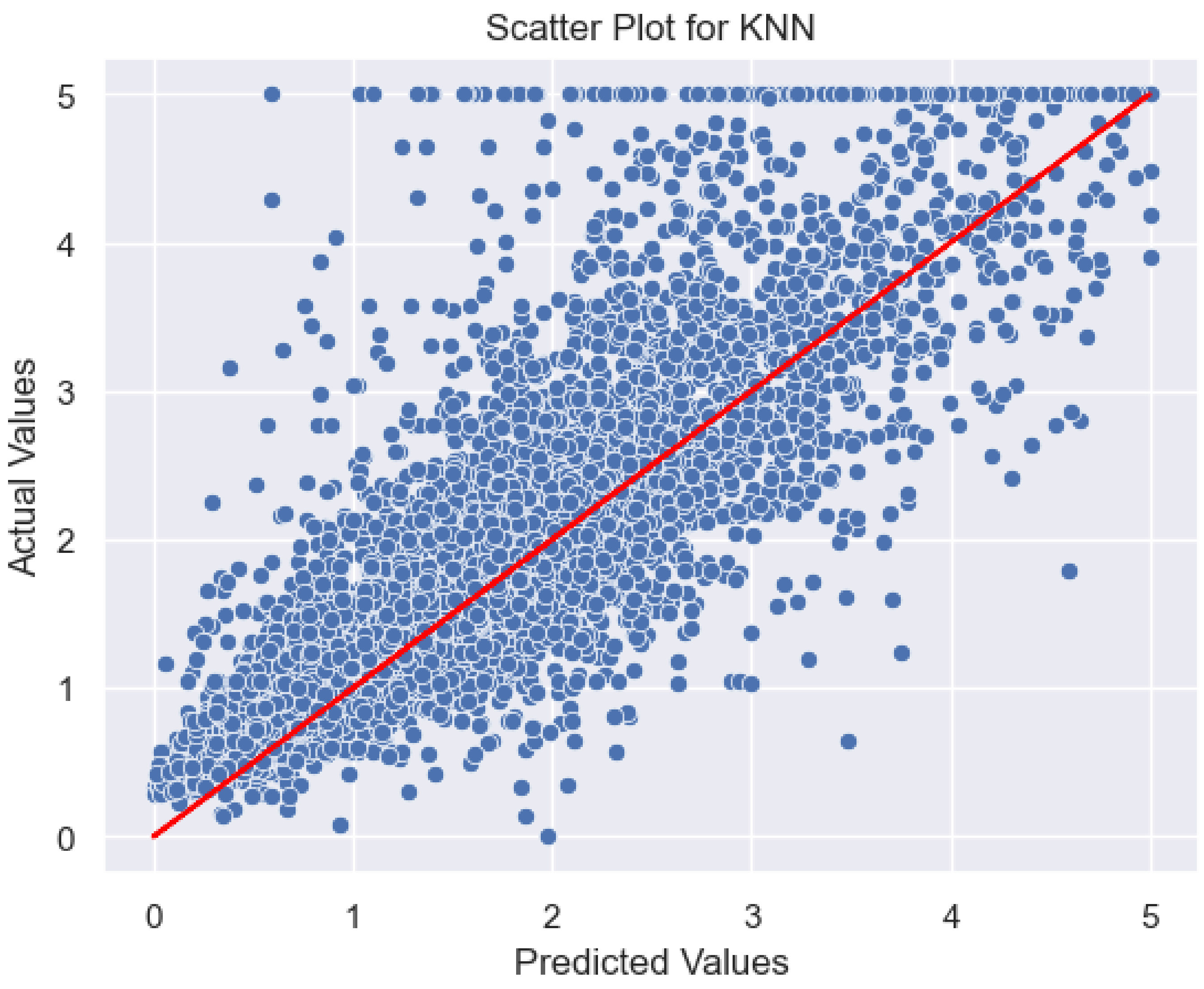
[86] ✓ 0.2s



```
knn.fit(X_train , y_train)
print('Training Score: ', knn.score(X_train, y_train))
print('predicted Score: ', knn.score(X_test, predicted_values))

```

[89] ✓ 1.4s

```
Training Score:  0.8145866287668804
predicted Score:  0.9601838301013791
```



Facial Expressions

Link in Kaggle:

https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data

Dataset information:

The data consists of 48x48 pixel grayscale images of faces. The faces have been automatically registered so that the face is more or less centered and occupies about the same amount of space in each image. The task is to categorize each face based on the emotion shown in the facial expression in to one of seven categories (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral).
train.csv contains two columns, "emotion" and "pixels". The "emotion" column contains a numeric code ranging from 0 to 6, inclusive, for the emotion that is present in the image. The "pixels" column contains a string surrounded in quotes for each image. The contents of this string a space-separated pixel values in row major order. test.csv contains only the "pixels" column and your task is to predict the emotion column.

The data shape was `(35887, 3)`

But we took a sample of the data

For each class only 1000 row , 500 training , 200 validation, 300 testing

Preprocessing and Feature Extraction:

The preprocessing phase had been done by several steps

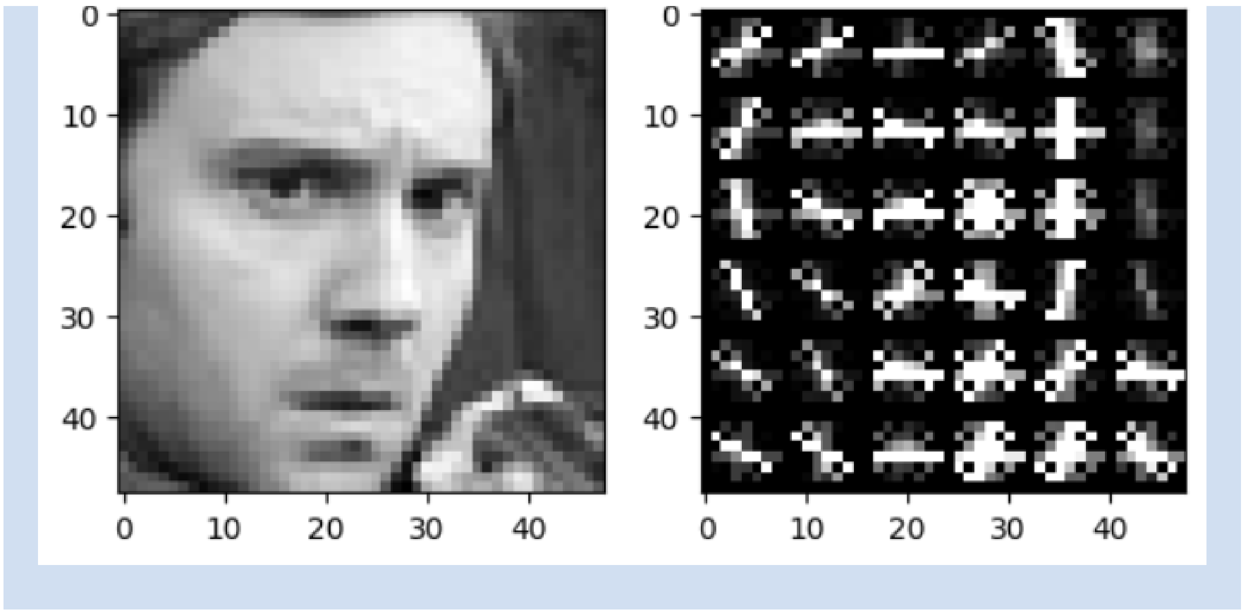
1. At first reshape the image to be of shape (48,48,1) and convert the type to nint8
2. Resize the image to be of size (64*128)
3. And then apply some methods to extract features

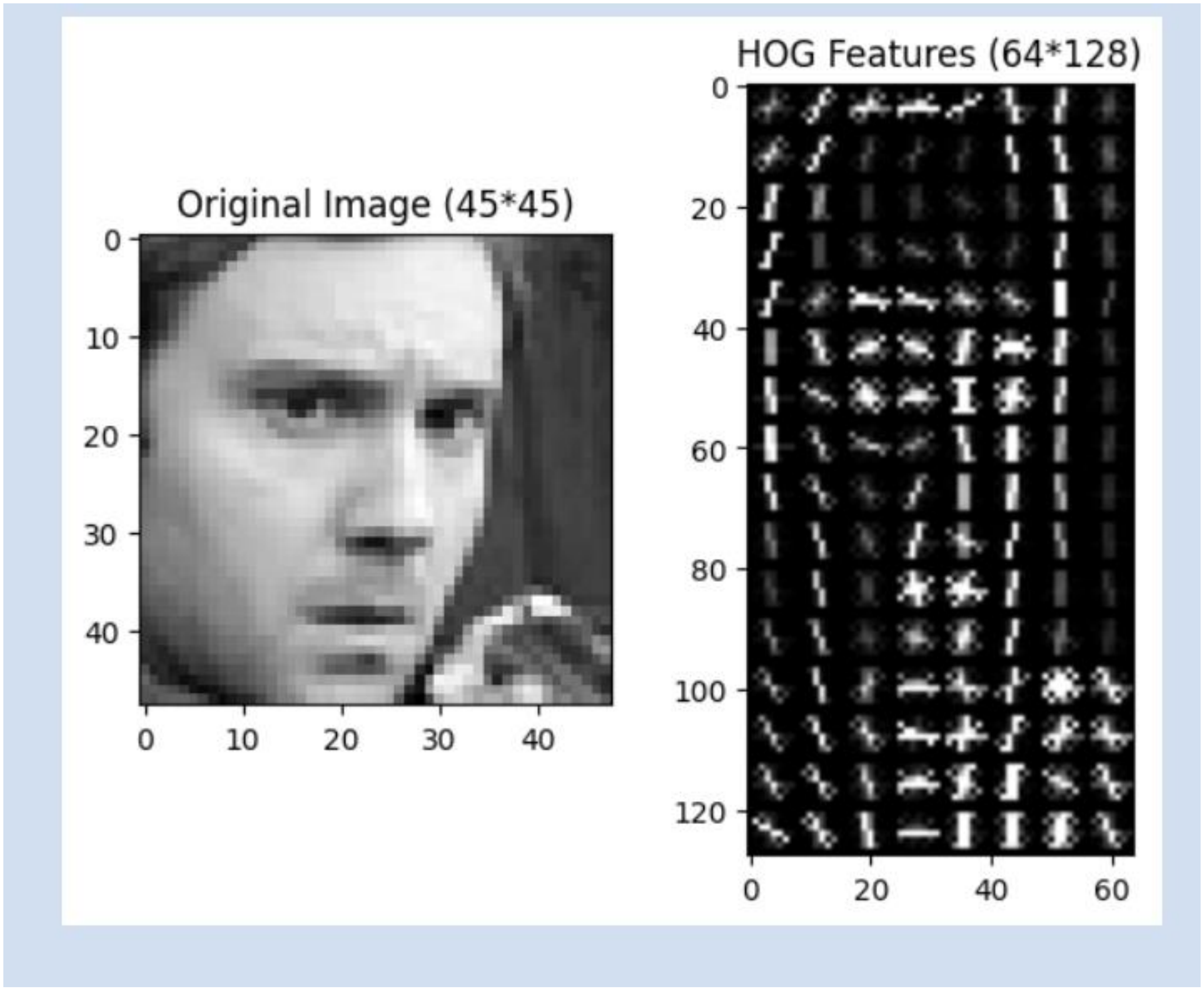
Feature Extraction ways

1.Hog features:

The image is divided into small cells, and for each cell, the gradient magnitude and orientation are computed. Histograms of gradient orientations are then created for each block of cells, and these histograms are concatenated to form the feature vector.

And the hog features had been applied to different sized image , to compare the accuracy



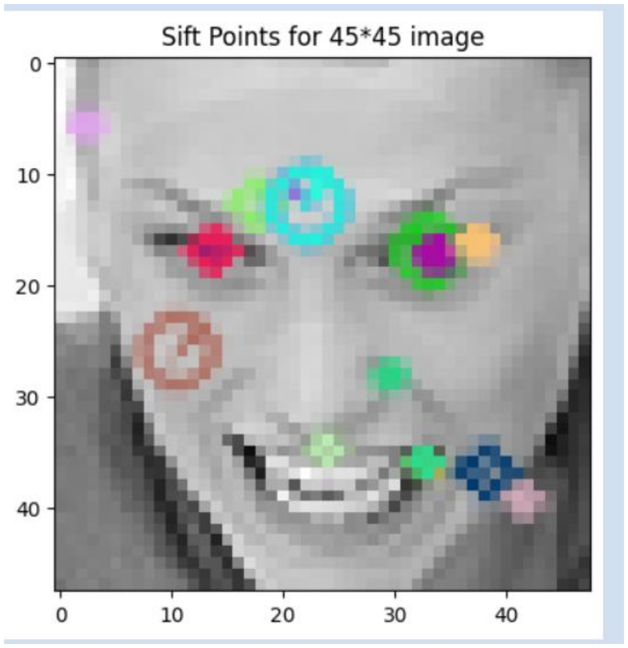


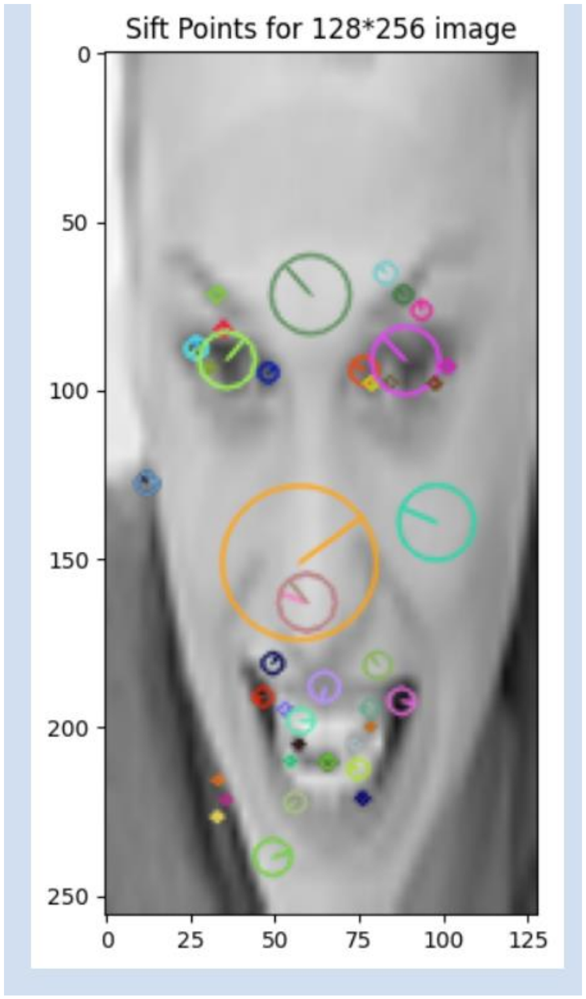
The result of extracting HOG features from an image is typically a one-dimensional feature vector for each region or block in the image. This feature vector represents the local gradients of the image and captures information about the shapes and textures present in that region.

2. Sift Algorithm:

SIFT is a keypoint detection and descriptor algorithm that is invariant to scale, rotation, and illumination changes.

Keypoints are identified in an image, and descriptors are computed based on the local image gradients in the regions surrounding the keypoints. SIFT features are designed to be robust to transformations and changes in viewpoint.

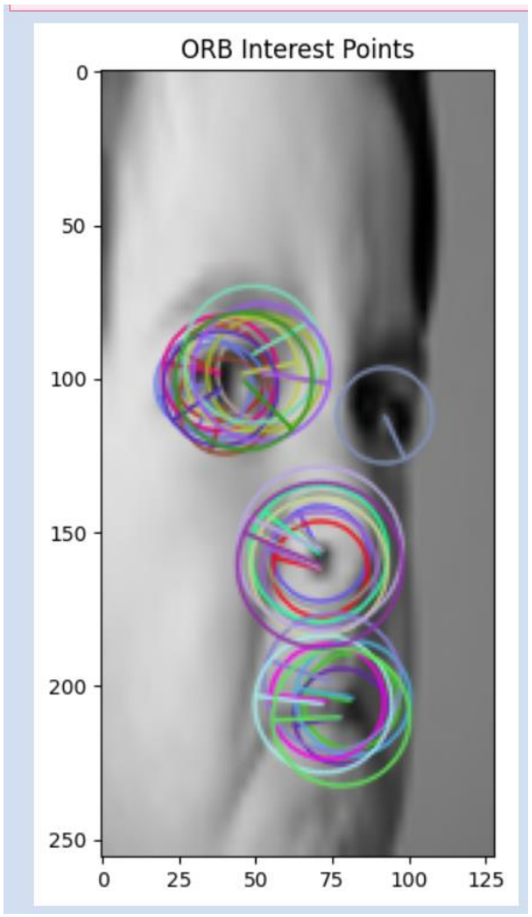




2.ORB:

ORB is a feature descriptor that combines the efficiency of the FAST (Features from Accelerated Segment Test) keypoint detector with the robustness of the BRIEF (Binary Robust Independent Elementary Features) descriptor.

Working Principle: ORB first identifies keypoints using the FAST algorithm and then computes binary descriptors using BRIEF. It introduces rotation invariance by considering the orientation of keypoints.

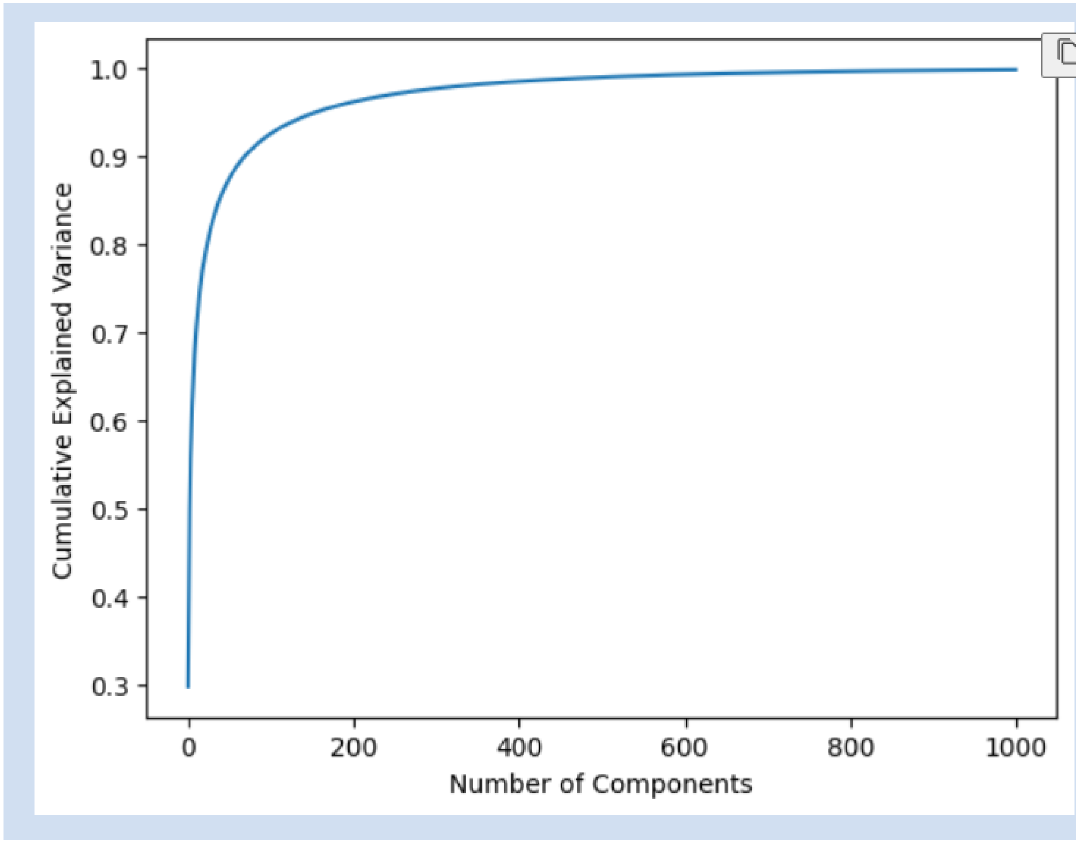


4.PCA:

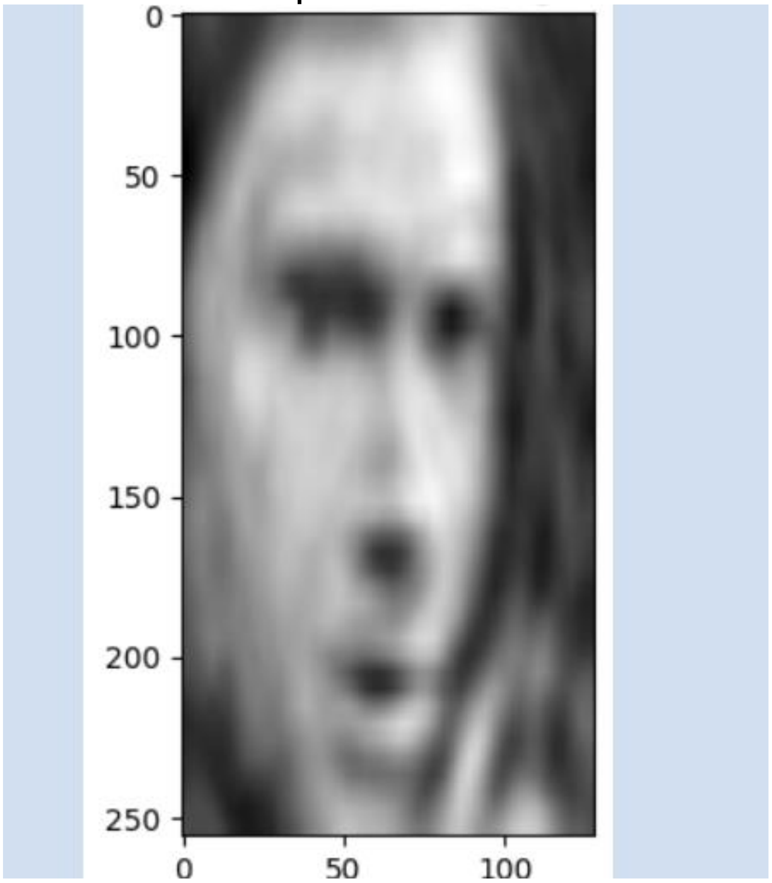
PCA is a dimensionality reduction technique that transforms high-dimensional data into a lower-dimensional representation by capturing the principal components or directions of maximum variance.

Working Principle: PCA identifies the eigenvectors and eigenvalues of the data covariance matrix and projects the data onto a reduced set of orthogonal axes.

We had used this graph to decide the number of components for pca



After applying the pca the number of columns had been reduced to the number of components , and each image represented by less number of pixels



Then the dataframes had been saved for applying the models

Logistic regression :

1.Overview

Logistic regression is a statistical method used for predicting the probability of a binary outcome based on one or more predictor variables. It's commonly used in machine learning for classification problems , but for our case we used a multiclass dataset ,
Multinomial logistic regression directly models the probability of each class.

2.Functionality

- **Predicts the probability of a binary outcome .**
- **Can be extended for multiclass classification.**
- **Estimates the relationship between the dependent variable and one or more independent variables.**

3. Usage

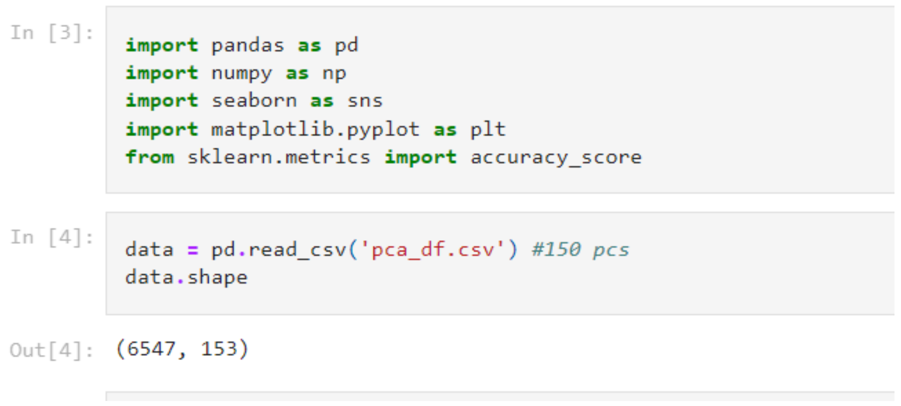
Logistic regression can be used in multiclass datasets by extending it using techniques like one-vs-rest or multinomial logistic regression. This allows logistic regression to handle classification problems with more than two classes by creating separate models for each class or directly modeling the probability of each class.

4. Parameters of model

penalty: 'l1', 'l2', or 'none'
C: inverse of regularization strength
solver: 'liblinear', 'newton-cg', 'lbfgs', 'sag', 'saga'

5. Implementation of Model

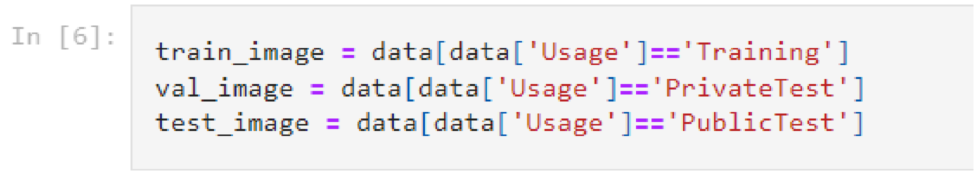
Based on our dataset (Facial Expressions) , that includes 7 distinct classes.



```
In [3]:  import pandas as pd
         import numpy as np
         import seaborn as sns
         import matplotlib.pyplot as plt
         from sklearn.metrics import accuracy_score

In [4]:  data = pd.read_csv('pca_df.csv') #150 pcs
         data.shape

Out[4]: (6547, 153)
```

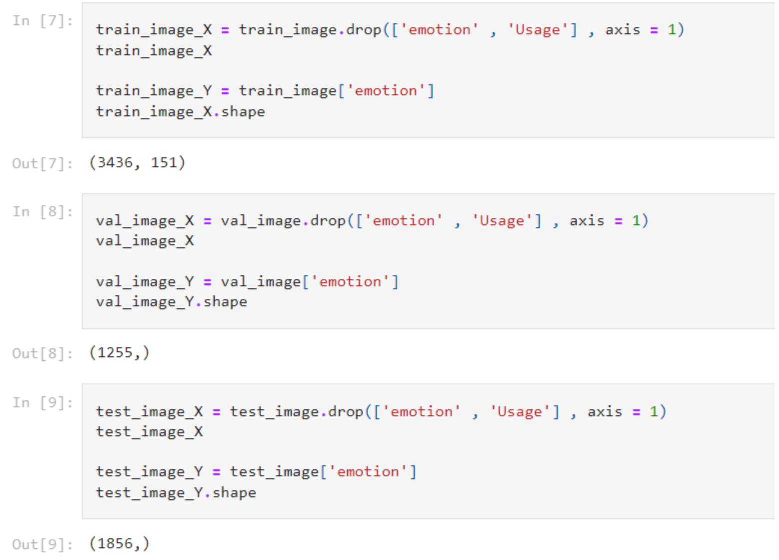
z

First importing used libraries , import image dataset.



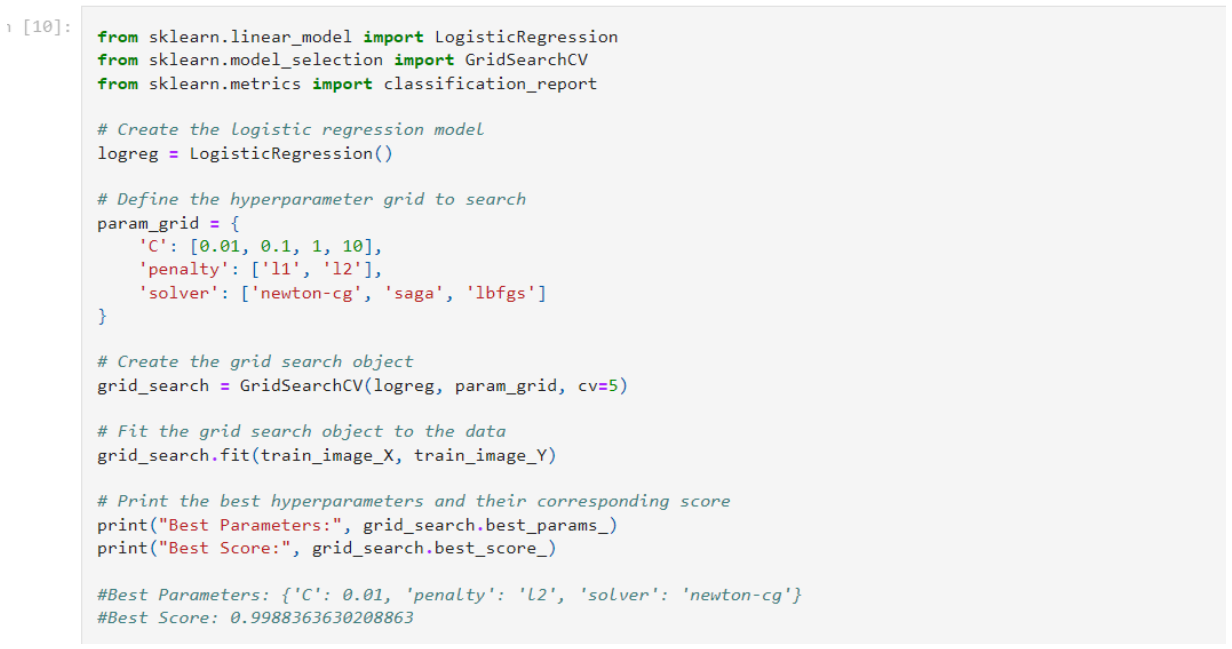
```
In [6]:  train_image = data[data['Usage']=='Training']
         val_image = data[data['Usage']=='PrivateTest']
         test_image = data[data['Usage']=='PublicTest']
```

Second we divide the dataset into train , validation , test set .



```
In [7]:  train_image_X = train_image.drop(['emotion' , 'Usage'] , axis = 1)
         train_image_X

         train_image_Y = train_image['emotion']
         train_image_X.shape

Out[7]:  (3436, 151)

In [8]:  val_image_X = val_image.drop(['emotion' , 'Usage'] , axis = 1)
         val_image_X

         val_image_Y = val_image['emotion']
         val_image_Y.shape

Out[8]:  (1255,)

In [9]:  test_image_X = test_image.drop(['emotion' , 'Usage'] , axis = 1)
         test_image_X

         test_image_Y = test_image['emotion']
         test_image_Y.shape

Out[9]:  (1856,)
```

Applying GridSearchCV

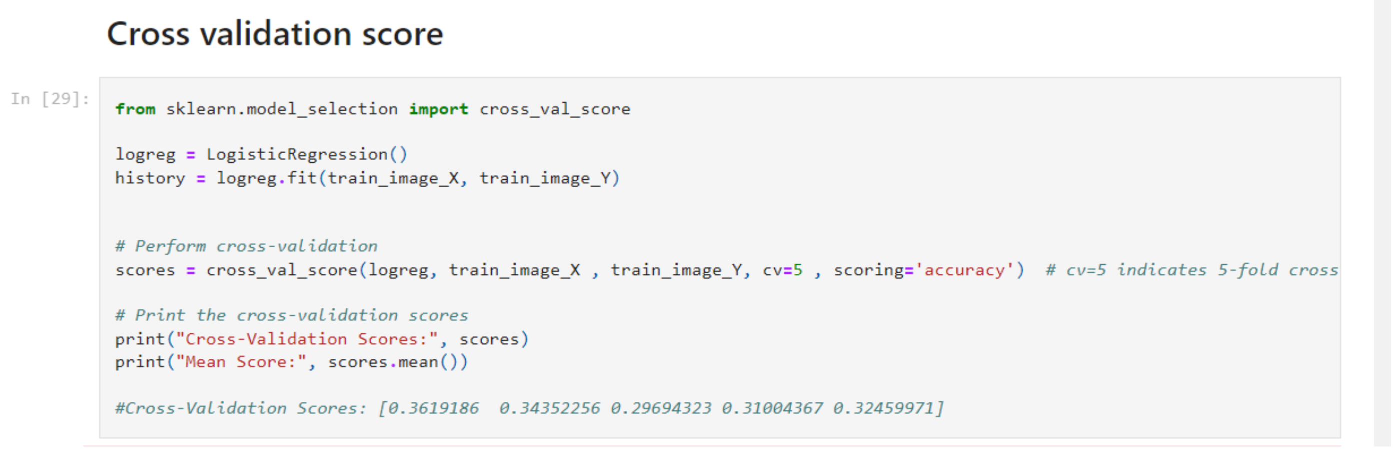


```
[10]:
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report

# Create the logistic regression model
logreg = LogisticRegression()

# Define the hyperparameter grid to search
param_grid = {
    'C': [0.01, 0.1, 1, 10],
    'penalty': ['l1', 'l2'],
    'solver': ['newton-cg', 'saga', 'lbfgs']
}

# Create the grid search object
grid_search = GridSearchCV(logreg, param_grid, cv=5)

# Fit the grid search object to the data
grid_search.fit(train_image_X, train_image_Y)

# Print the best hyperparameters and their corresponding score
print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

#Best Parameters: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
#Best Score: 0.9988363630208863
```

GridSearchCV can be used with logistic regression to tune hyperparameters and find the best model for a given dataset.
It systematically tests a grid of hyperparameter values and performs cross-validation to determine the optimal combination(best parameters with best score). This can help improve the performance of logistic regression on a specific task.
The produced parameters for best score

```
Best Parameters: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
Best Score: 0.9988363630208863
```

But by using these parameters leads to very low test accuracy
So we are going to make Cross validation test to determine the accuracy can produced from the Validation set



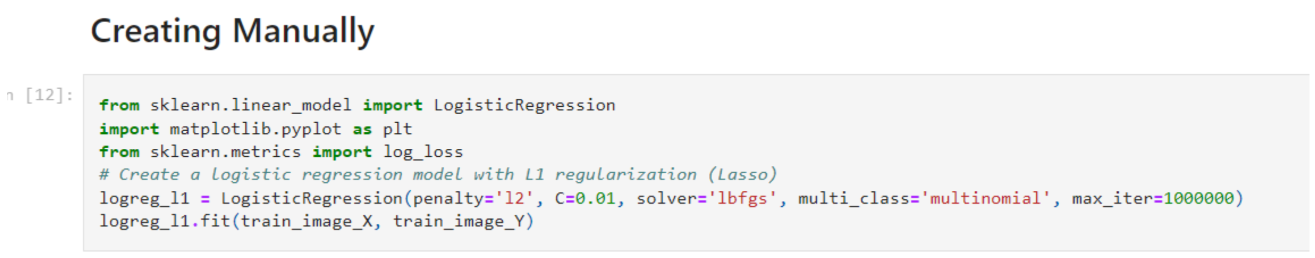
Cross validation score

```
In [29]:
from sklearn.model_selection import cross_val_score

logreg = LogisticRegression()
history = logreg.fit(train_image_X, train_image_Y)


# Perform cross-validation
scores = cross_val_score(logreg, train_image_X , train_image_Y, cv=5 , scoring='accuracy')  # cv=5 indicates 5-fold cross

# Print the cross-validation scores
print("Cross-Validation Scores:", scores)
print("Mean Score:", scores.mean())

#Cross-Validation Scores: [0.3619186  0.34352256 0.29694323 0.31004367 0.32459971]
```

The result of Cross Validation is

```
Cross-Validation Scores: [0.3619186  0.34352256 0.29694323 0.31149927 0.32459
971]
Mean Score: 0.3276966758065062
```

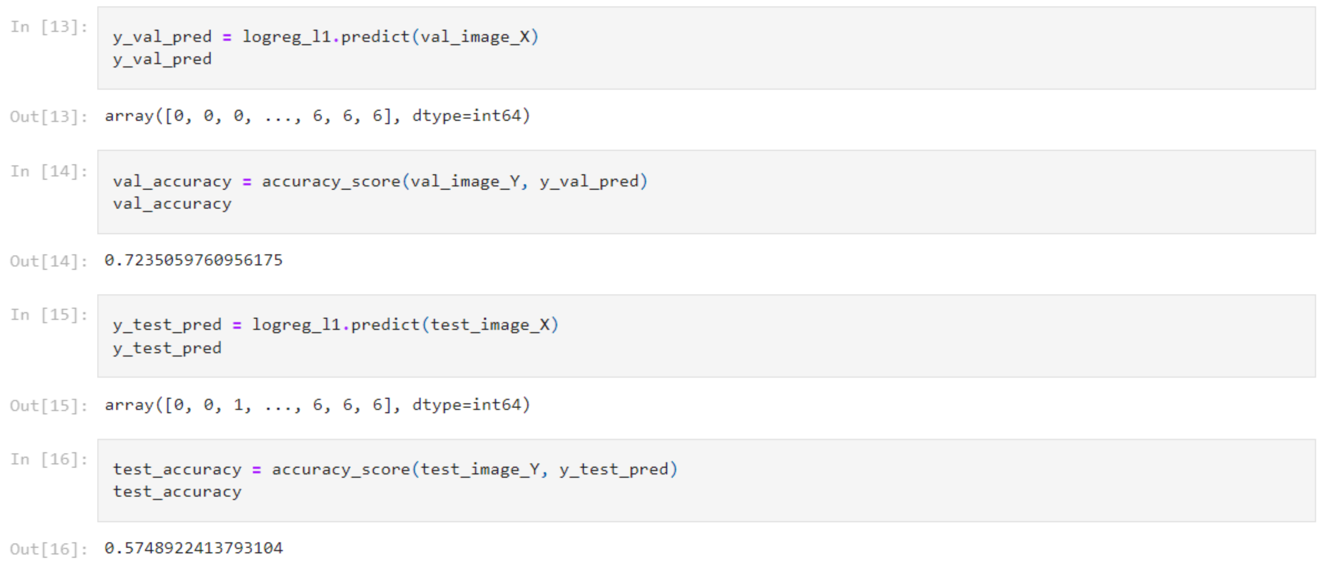
As the result of accuracy of cross validation is too small , therefore the test set will not be efficient and its accuracy will be very small
So we are going to fit the model parameters manually for finding hyperparameters that leads the model for high accuracy validation and test sets .



Creating Manually

```
n [12]:
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt
from sklearn.metrics import log_loss
# Create a logistic regression model with L1 regularization (Lasso)
logreg_l1 = LogisticRegression(penalty='l2', C=0.01, solver='lbfgs', multi_class='multinomial', max_iter=1000000)
logreg_l1.fit(train_image_X, train_image_Y)
```

By using this hyperparameters , here is the accuracy we gets after predicting on y Validation and test

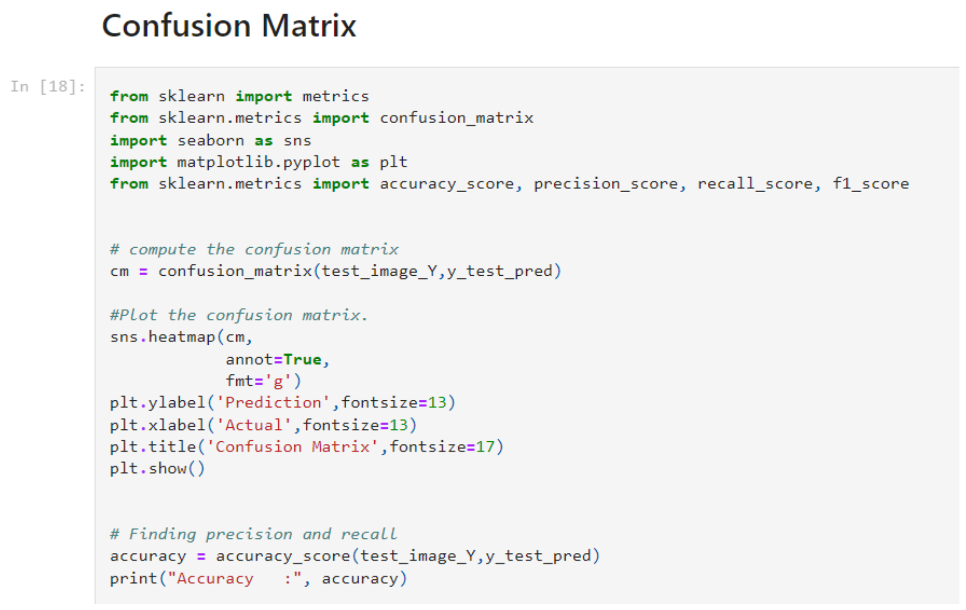


```
In [13]: y_val_pred = logreg_l1.predict(val_image_X)
         y_val_pred

Out[13]: array([0, 0, 0, ..., 6, 6, 6], dtype=int64)

In [14]: val_accuracy = accuracy_score(val_image_Y, y_val_pred)
         val_accuracy

Out[14]: 0.7235059760956175

In [15]: y_test_pred = logreg_l1.predict(test_image_X)
         y_test_pred

Out[15]: array([0, 0, 1, ..., 6, 6, 6], dtype=int64)

In [16]: test_accuracy = accuracy_score(test_image_Y, y_test_pred)
         test_accuracy

Out[16]: 0.5748922413793104
```

The best validation accuracy can get : 72%
The best test accuracy can get : 57%

Applying Confusion matrix

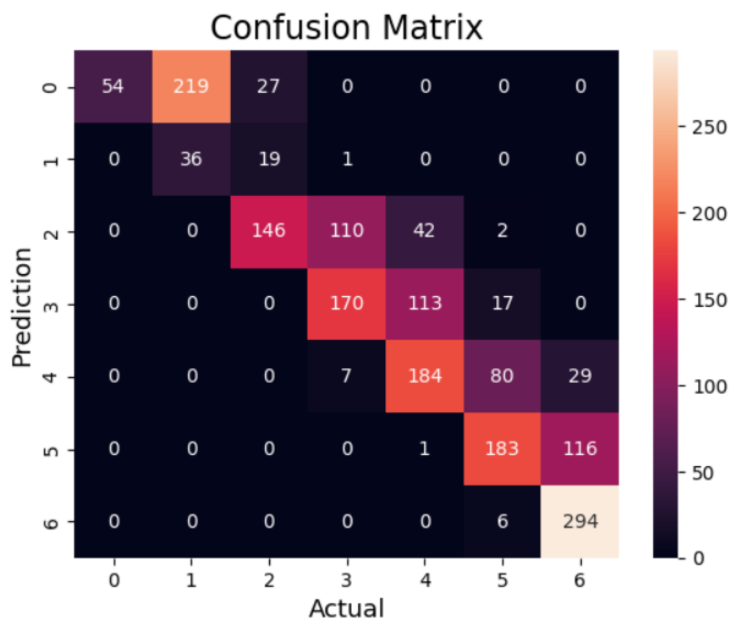
the purpose of applying a confusion matrix on logistic regression for multiclass classification is to evaluate the model's performance by providing a detailed breakdown of correct and incorrect predictions for each class.



Confusion Matrix

```
In [18]: from sklearn import metrics
         from sklearn.metrics import confusion_matrix
         import seaborn as sns
         import matplotlib.pyplot as plt
         from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score


         # compute the confusion matrix
         cm = confusion_matrix(test_image_Y,y_test_pred)

         #Plot the confusion matrix.
         sns.heatmap(cm,
                     annot=True,
                     fmt='g')
         plt.ylabel('Prediction',fontsize=13)
         plt.xlabel('Actual',fontsize=13)
         plt.title('Confusion Matrix',fontsize=17)
         plt.show()


         # Finding precision and recall
         accuracy = accuracy_score(test_image_Y,y_test_pred)
         print("Accuracy   :", accuracy)
```

The result of test accuracy



Prediction \ Actual	0	1	2	3	4	5	6
0	54	219	27	0	0	0	0
1	0	36	19	1	0	0	0
2	0	0	146	110	42	2	0
3	0	0	0	170	113	17	0
4	0	0	0	7	184	80	29
5	0	0	0	0	1	183	116
6	0	0	0	0	0	6	294

```
Accuracy    : 0.5748922413793104
```

Plotting ROC

(The Reciever operating characteristic curve plots the true positive (TP) rate versus the false positive (FP) rate at different classification thresholds)

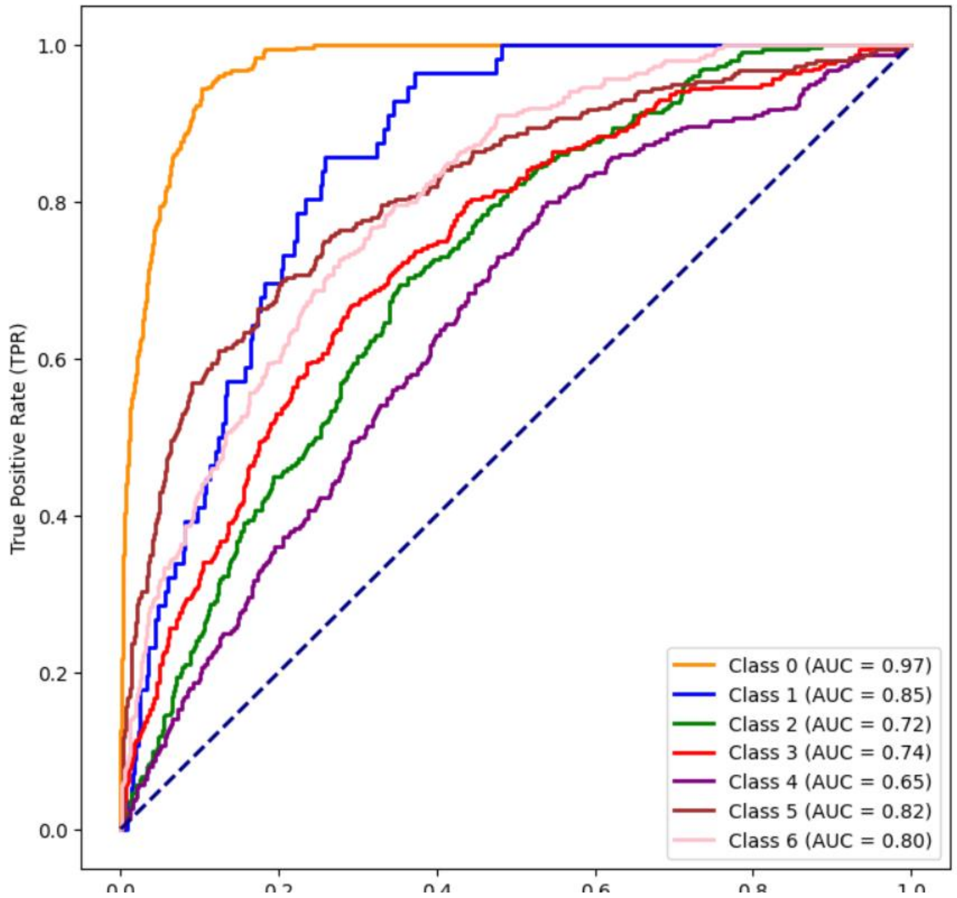
In [22]:
```
y_probs = logreg.predict_proba(test_image_X)

fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(logreg.classes_)):
    fpr[i], tpr[i], _ = roc_curve(test_image_Y == logreg.classes_[i], y_probs[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

plt.figure(figsize=(8, 8))
colors = ['darkorange', 'blue', 'green', 'red', 'purple', 'brown', 'pink']
for i in range(len(logreg.classes_)):
    plt.plot(fpr[i], tpr[i], color=colors[i], lw=2, label=f'Class {logreg.classes_[i]} (AUC = {roc_auc[i]:.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve - Multi-Class')
plt.legend(loc='lower right')
plt.show()
```

1. **"fpr = dict()", "tpr = dict()", and "roc_auc = dict()" indicate the initialization of three dictionaries in Python. These dictionaries are likely intended to store false positive rates (fpr), true positive rates (tpr), and ROC AUC (Area Under the Curve) values for different classes or thresholds, which are common components used in evaluating the performance of classification models, particularly in the context of ROC curve analysis.**

2. **It iterates over each class in the logistic regression model to calculate the false positive rate (fpr), true positive rate (tpr), and ROC AUC (Area Under the Curve) for each class.**

It uses the roc_curve function to compute the fpr and tpr based on the predicted probabilities (y_probs) and the actual classes (test_image_Y). The auc function is then used to calculate the ROC AUC for each class based on the fpr and tpr values.

Result

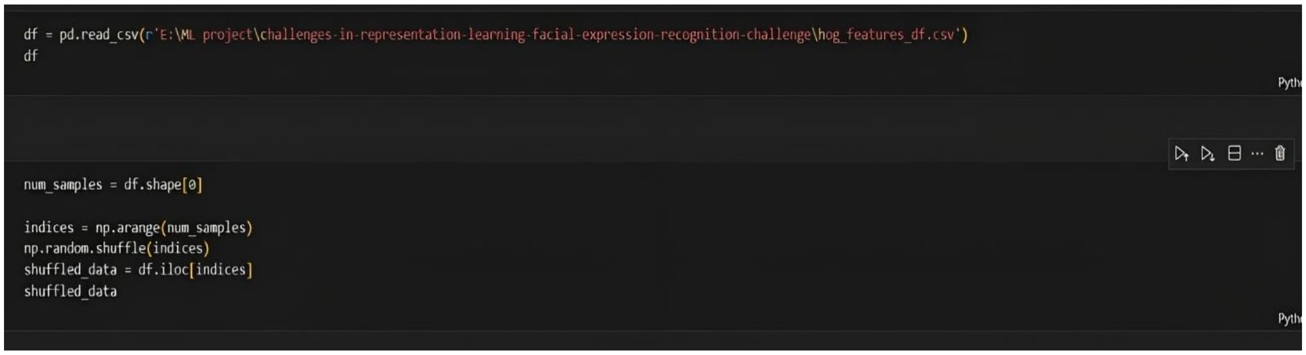


Kmeans clustering

K-means is a popular clustering algorithm used to partition a dataset into distinct, non-overlapping groups or clusters. The algorithm aims to minimize the within-cluster variance, meaning that data points within the same cluster are as similar as possible. In the context of facial expression analysis, this means grouping similar facial expressions together.

Implementation

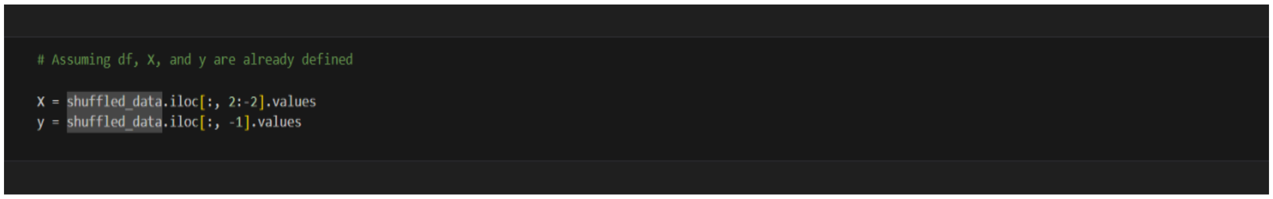
step 1: Shuffles the dataset to ensure randomness in training.



```
df = pd.read_csv(r'E:\ML project\challenges-in-representation-learning-facial-expression-recognition-challenge\hog_features_df.csv')
df
```

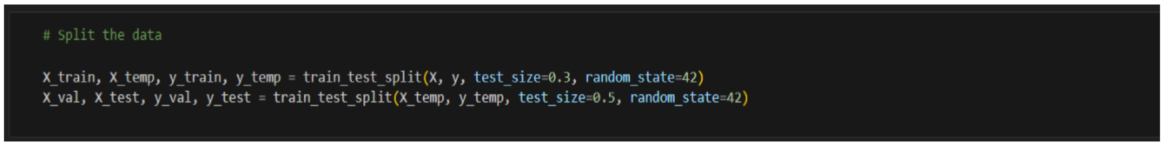
```
num_samples = df.shape[0]

indices = np.arange(num_samples)
np.random.shuffle(indices)
shuffled_data = df.iloc[indices]
shuffled_data
```

step 2: X contains the values of all feature , while y contains the values from the 'emotions' column.



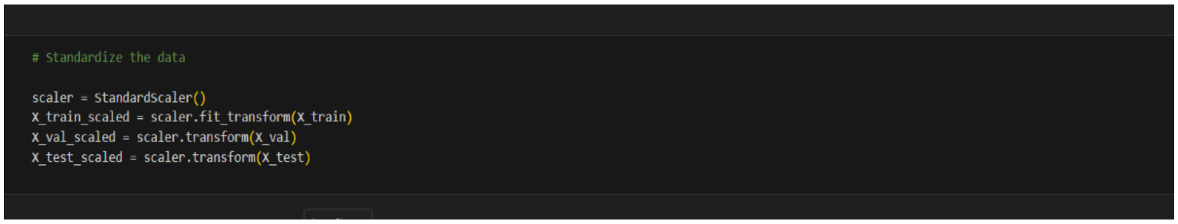
```
# Assuming df, X, and y are already defined

X = shuffled_data.iloc[:, 2:-2].values
y = shuffled_data.iloc[:, -1].values
```

step 3: split data to :
70% : training (X_train, y_train)
15% : validation(X_val, y_val)
15% : testing(X_test, y_test)



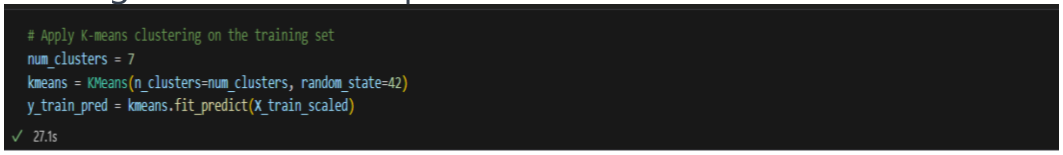
```
# Split the data

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

Step 4 : standardizing the data using the `StandardScaler` from scikit-learn. Standardization is a common preprocessing step in machine learning that transforms the features to have a mean of 0 and a standard deviation of 1. This process is crucial for certain algorithms, particularly those that rely on distance measures between data points, such as k-means clustering.



```
# Standardize the data

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
```

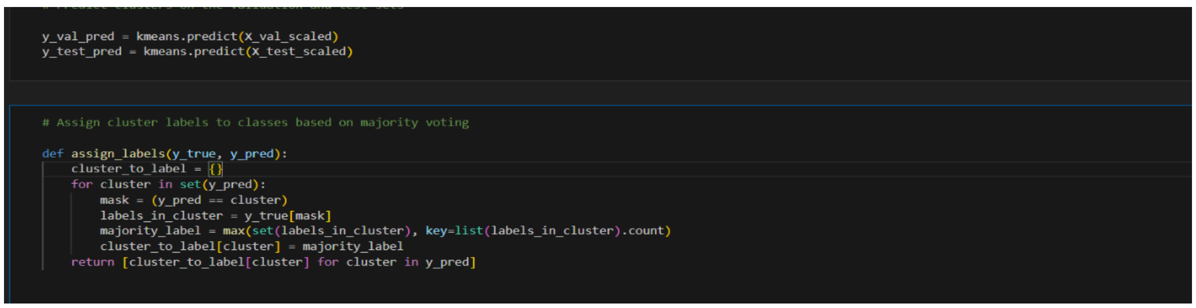
Step 5: Applies the k-means clustering algorithm on the standardized training set and obtains predicted cluster labels.



```
# Apply K-means clustering on the training set
num_clusters = 7
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
y_train_pred = kmeans.fit_predict(X_train_scaled)
✓ 27.1s
```

Step 6:

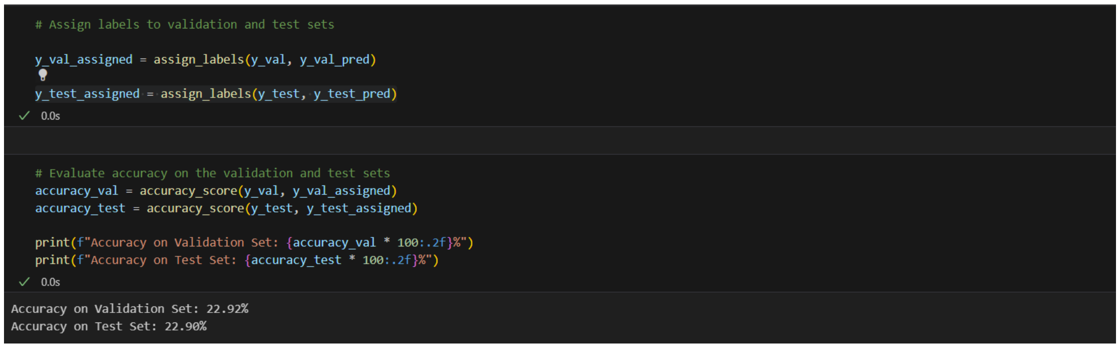
- Predicts clusters for the validation and test sets using the trained k-means model.
- The purpose of this function is to assign meaningful class labels to the clusters predicted by an unsupervised learning algorithm. Since the algorithm itself does not know the true class labels, the majority voting strategy is employed to associate each cluster with the class label that is most prevalent within that cluster. This allows for a more interpretable analysis of the clustering results, especially when there is an interest in understanding the characteristics of each cluster in terms of the original class labels.



```
y_val_pred = kmeans.predict(X_val_scaled)
y_test_pred = kmeans.predict(X_test_scaled)
```

```
# Assign cluster labels to classes based on majority voting

def assign_labels(y_true, y_pred):
    cluster_to_label = {}
    for cluster in set(y_pred):
        mask = (y_pred == cluster)
        labels_in_cluster = y_true[mask]
        majority_label = max(set(labels_in_cluster), key=list(labels_in_cluster).count)
        cluster_to_label[cluster] = majority_label
    return [cluster_to_label[cluster] for cluster in y_pred]
```

Step 7: Evaluates the accuracy of the clustering results on the validation and test sets.

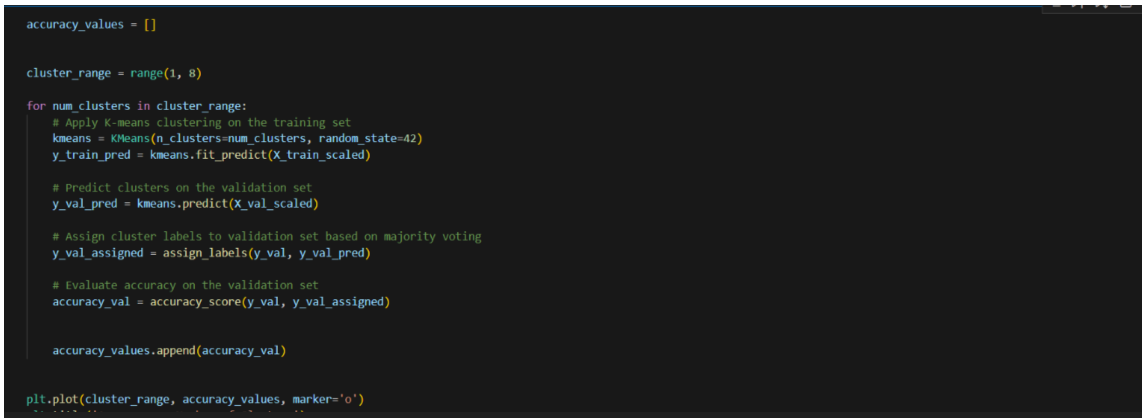


```
# Assign labels to validation and test sets

y_val_assigned = assign_labels(y_val, y_val_pred)

y_test_assigned = assign_labels(y_test, y_test_pred)
```
✓ 0.0s

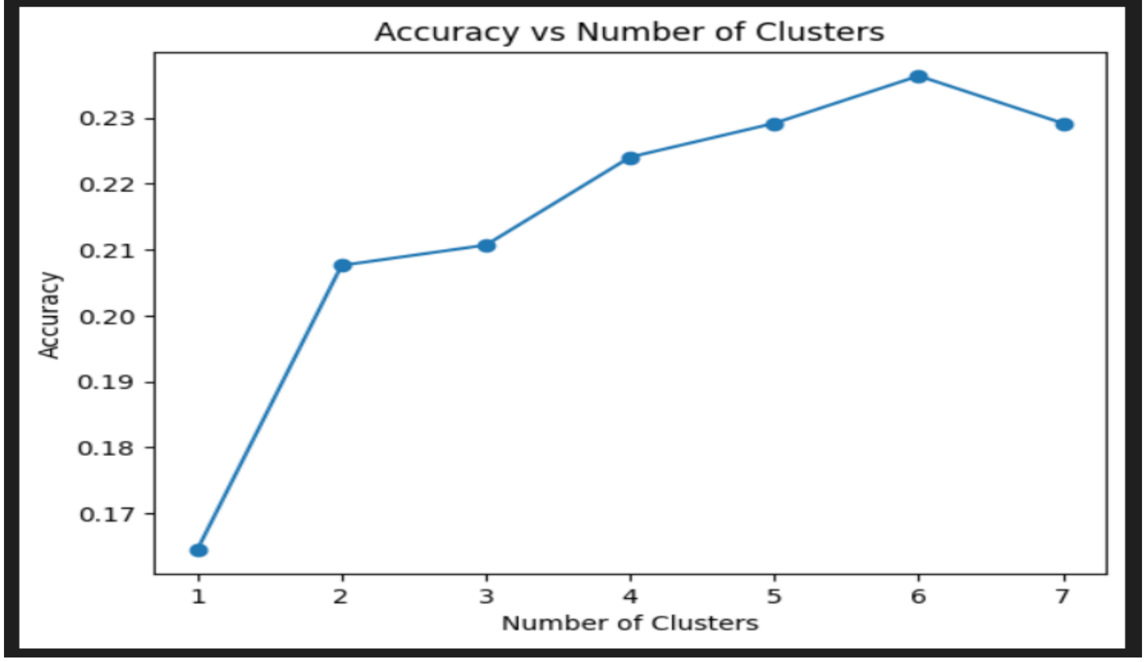
```
# Evaluate accuracy on the validation and test sets
accuracy_val = accuracy_score(y_val, y_val_assigned)
accuracy_test = accuracy_score(y_test, y_test_assigned)

print(f"Accuracy on Validation Set: {accuracy_val * 100:.2f}%")
print(f"Accuracy on Test Set: {accuracy_test * 100:.2f}%")
```
✓ 0.0s

```
Accuracy on Validation Set: 22.92%
Accuracy on Test Set: 22.90%
```

Step 8 : Explores how the accuracy varies with different numbers of clusters using a line plot.



```
accuracy_values = []


cluster_range = range(1, 8)

for num_clusters in cluster_range:
    # Apply K-means clustering on the training set
    kmeans = KMeans(n_clusters=num_clusters, random_state=42)
    y_train_pred = kmeans.fit_predict(X_train_scaled)

    # Predict clusters on the validation set
    y_val_pred = kmeans.predict(X_val_scaled)

    # Assign cluster labels to validation set based on majority voting
    y_val_assigned = assign_labels(y_val, y_val_pred)

    # Evaluate accuracy on the validation set
    accuracy_val = accuracy_score(y_val, y_val_assigned)


    accuracy_values.append(accuracy_val)


plt.plot(cluster_range, accuracy_values, marker='o')
```

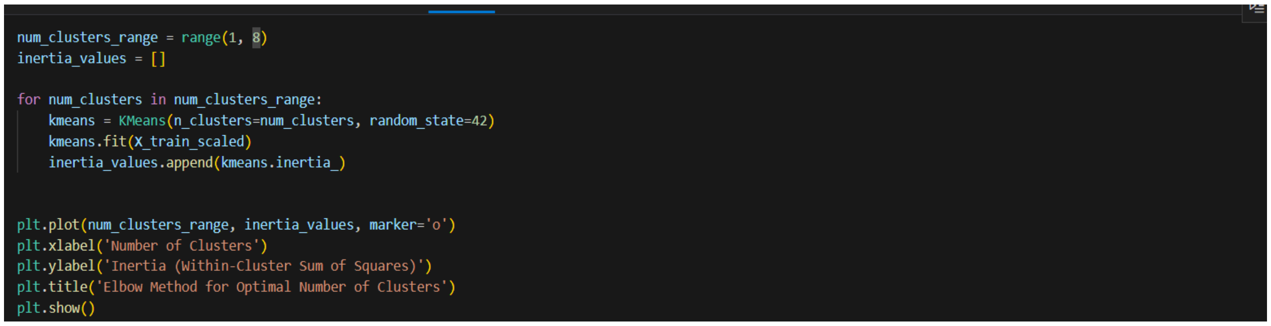


Step 9 :

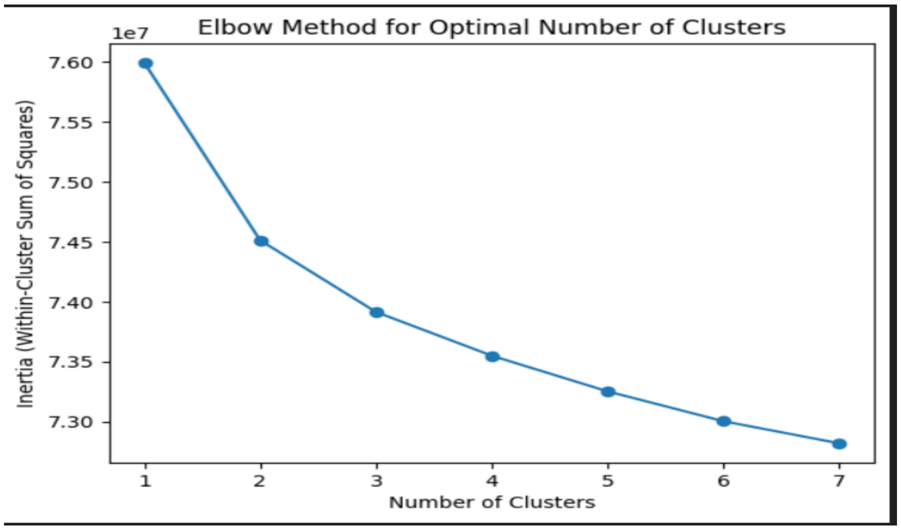
- The Elbow Method is a technique used to determine the optimal number of clusters in a dataset for K-means clustering. The idea is to plot the explained variation as a function of the number of clusters and look for an "elbow" point, where adding more clusters does not significantly improve the model's performance. Here's a detailed explanation of the Elbow Method:

- K-means Inertia:

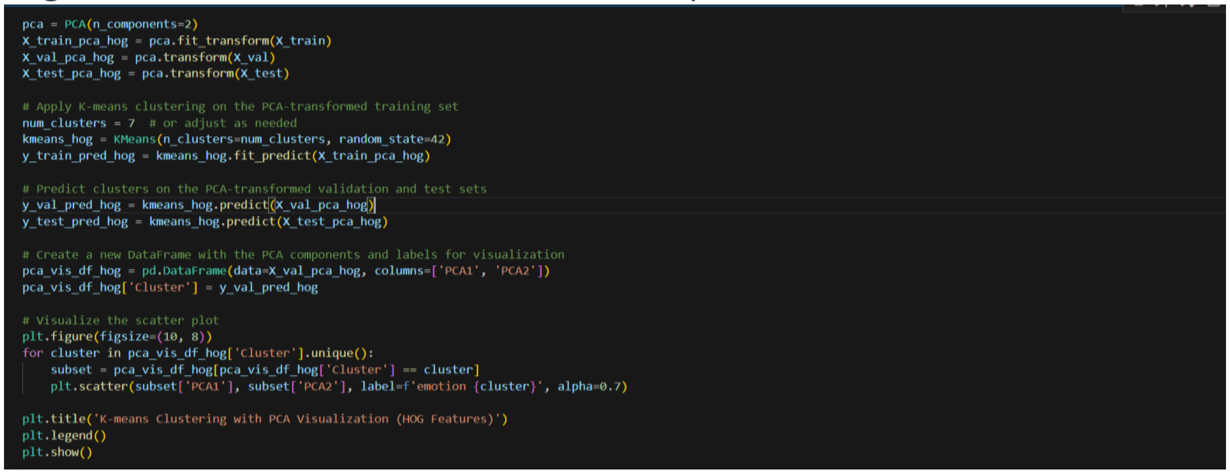
In K-means clustering, "inertia" or "within-cluster sum of squares" is a metric that measures how far each data point in a cluster is from the center of that cluster. It is calculated as the sum of squared distances from each point to its assigned cluster center.



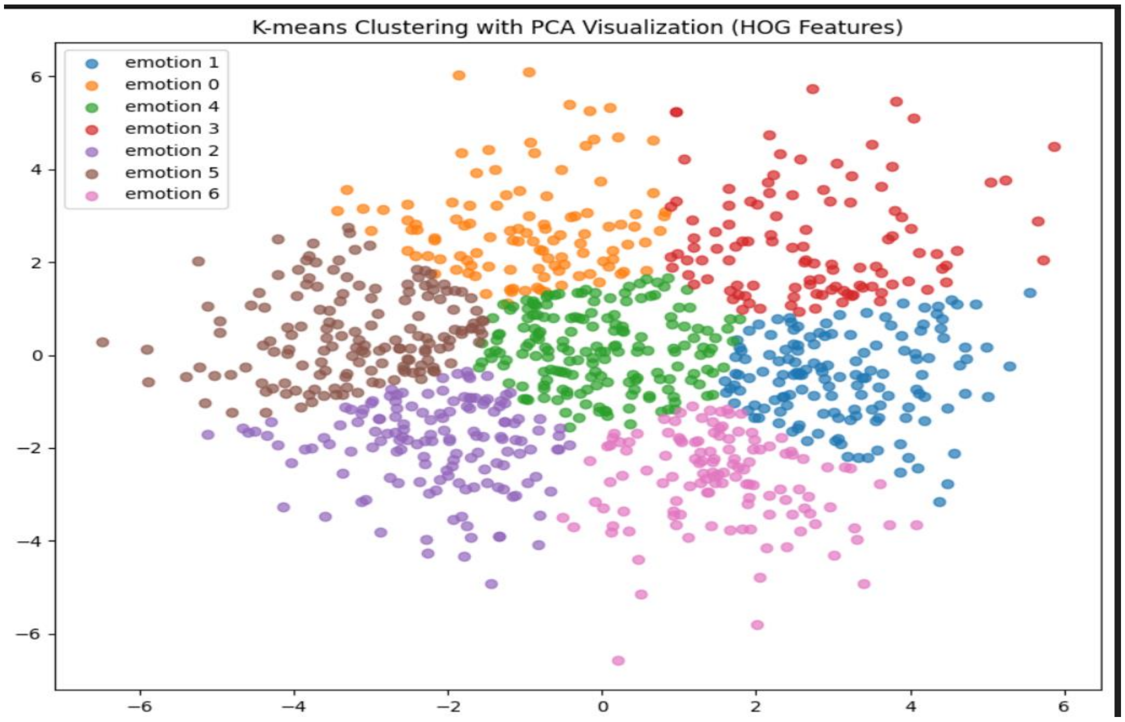
```python
num_clusters_range = range(1, 8)
inertia_values = []

for num_clusters in num_clusters_range:
    kmeans = KMeans(n_clusters=num_clusters, random_state=42)
    kmeans.fit(X_train_scaled)
    inertia_values.append(kmeans.inertia_)


plt.plot(num_clusters_range, inertia_values, marker='o')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia (Within-Cluster Sum of Squares)')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.show()
```



Step 10 : The purpose of this code is to visually explore and understand the clustering results obtained from applying K-means on the reduced 2-dimensional feature space obtained through PCA. The scatter plot provides insights into how well the algorithm has grouped similar facial expressions together based on the reduced feature representation



```python
pca = PCA(n_components=2)
X_train_pca_hog = pca.fit_transform(X_train)
X_val_pca_hog = pca.transform(X_val)
X_test_pca_hog = pca.transform(X_test)

# Apply K-means clustering on the PCA-transformed training set
num_clusters = 7  # or adjust as needed
kmeans_hog = KMeans(n_clusters=num_clusters, random_state=42)
y_train_pred_hog = kmeans_hog.fit_predict(X_train_pca_hog)

# Predict clusters on the PCA-transformed validation and test sets
y_val_pred_hog = kmeans_hog.predict(X_val_pca_hog)
y_test_pred_hog = kmeans_hog.predict(X_test_pca_hog)

# Create a new DataFrame with the PCA components and labels for visualization
pca_vis_df_hog = pd.DataFrame(data=X_val_pca_hog, columns=['PCA1', 'PCA2'])
pca_vis_df_hog['Cluster'] = y_val_pred_hog

# Visualize the scatter plot
plt.figure(figsize=(10, 8))
for cluster in pca_vis_df_hog['Cluster'].unique():
    subset = pca_vis_df_hog[pca_vis_df_hog['Cluster'] == cluster]
    plt.scatter(subset['PCA1'], subset['PCA2'], label=f'emotion {cluster}', alpha=0.7)

plt.title('K-means Clustering with PCA Visualization (HOG Features)')
plt.legend()
plt.show()
```


K-means Clustering with PCA Visualization (HOG Features)
emotion 1
emotion 0
emotion 4
emotion 3
emotion 2
emotion 5
emotion 6