Introduction:

The Project demonstrates how to use Python and scikit-learn, a popular machine learning library, to build and evaluate a k-nearest neighbors (KNN) classifier. KNN is a popular machine learning algorithm used for both classification and regression problems. The code begins by creating a synthetic dataset using scikit-learn's make_blobs function, which generates a dataset of points clustered in three different groups. The data is then split into a training set and a test set using scikit-learn's train_test_split function. The KNN classifier is then trained on the training data using the KNeighborsClassifier class, and its performance is evaluated on the training set. The code also demonstrates how to create a scatter plot of the data and visualize the decision boundary of the classifier. Finally, the code shows how to adjust the parameters of the KNN classifier to achieve different performance characteristics. Overall, this code provides a useful example of how to use scikit-learn to build and evaluate a KNN classifier, and how to visualize the results of the classifier on a synthetic dataset.

Import modules for this project

```
[9]  from sklearn import datasets
     from sklearn.metrics import accuracy_score
     from sklearn.model_selection import train_test_split
     import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     from sklearn.metrics import confusion_matrix, roc_curve, auc
     from sklearn.model_selection import train_test_split
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.datasets import make_blobs
```

The necessary modules are imported for this project. sklearn is a machine learning library for Python, and it includes modules for data processing modeling, and evaluation. pandas and numpy are additional libraries for data manipulation and analysis. matplotlib is a visualization library for creating plots and graphs. make_blobs function is used to generate simulated dataset.

Data:

Create a simulated dataset

```
centers = [[2, 4], [6, 6], [1, 9]]
n_classes = len(centers)
data, labels = make_blobs(n_samples=150,
                          centers=np.array(centers),
                          random_state=1)
```

The above code snippet creates a synthetic dataset consisting of 150 records. The dataset is generated such that there are three clusters, each with a different center. The centers of the three clusters are defined by the centers list, which contains three points in two-dimensional space:

[2, 4], [6, 6], and [1, 9]. The n_classes variable is set to 3, which is the number of clusters in the synthetic dataset.

The make_blobs function generates the dataset by randomly sampling points from a Gaussian distribution centered at each of the three cluster centers. The standard deviation of each cluster is set to 1.0 by default. The resulting dataset is returned as two arrays: data, which contains the two-dimensional feature vectors for each sample, and labels, which contains the cluster assignments for each sample. The centers argument to make_blobs specifies the coordinates of the cluster centers as a two-dimensional NumPy array. The random_state argument is set to 1 to ensure the reproducibility of the dataset.

Split the data into Train and Test(80:20)

0s

```
res = train_test_split(data, labels,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=14)
train_data, test_data, train_labels, test_labels = res
```

The above code snippet uses the train_test_split function from the sklearn.model_selection module to split the synthetic dataset into training and testing subsets. The train_test_split function takes as input the dataset (data) and the corresponding labels (labels), and splits them into separate subsets for training and testing. The train_size and test_size arguments specify the proportion of the dataset to use for training and testing, respectively. In this case, 80% of the data is used for training, and 20% is used for testing.

The random_state argument sets the random seed, which ensures that the same split is generated every time the code is run. This is important for reproducibility, so that the same results can be obtained every time the code is run.

The train_test_split function returns a tuple of four arrays: train_data, test_data, train_labels, and test_labels. The train_data and test_data arrays contain the feature vectors for the training and testing subsets, respectively. The train_labels and test_labels arrays contain the corresponding labels for the training and testing subsets, respectively. These arrays are assigned to the variables train_data, test_data, train_labels, and test_labels, respectively, for later use in training and evaluating a machine learning model.

0s

```
# Create and fit a nearest-neighbor classifier
from sklearn.neighbors import KNeighborsClassifier
# classifier "out of the box", no parameters
knn = KNeighborsClassifier()
knn.fit(train_data, train_labels)
```

```
KNeighborsClassifier()
```

This code uses the KNeighborsClassifier class from the sklearn.neighbors module to create a nearest-neighbor classifier. The KNeighborsClassifier class is an implementation of the k-nearest neighbors algorithm, which is a type of instance-based learning that uses a distance metric to identify the k-closest samples in the training data to a given test sample, and then predicts the label of the test sample based on the majority class of its k nearest neighbors. The knn object is initialized as an instance of the KNeighborsClassifier class with no parameters specified. This means that the default values of the parameters are used, which include using k=5 and the Euclidean distance metric to calculate distances between samples. The fit method of the knn object is then called to train the nearest-neighbor classifier on the training data and labels. This involves storing the training data and labels in memory so that they can be used to make predictions on new, unseen data. Once the classifier has been trained, it can be used to make predictions on the test data to evaluate its performance.

```
# print some interested metrics
print("Predictions from the classifier:")
learn_data_predicted = knn.predict(train_data)
print(learn_data_predicted)
print("Target values:")
print(train_labels)
print(accuracy_score(learn_data_predicted, train_labels))
```

The above code snippet prints out some evaluation metrics for the trained nearest-neighbor classifier. The predict method of the knn object is used to generate predictions for the training data, which are stored in the learn_data_predicted variable. The code then prints out the predicted labels for the training data (learn_data_predicted) and the true labels (train_labels). This allows us to visually inspect how well the classifier is doing on the training data. Finally, the code computes the accuracy of the classifier on the training data using the accuracy_score functio. The accuracy_score function takes as input the predicted labels and the true labels, and returns the proportion of samples for which the predicted label matches the true label.

Accuracy is 1.0

```
# KNN using some specific parameters.
knn2 = KNeighborsClassifier(algorithm='auto',
                            leaf_size=30,
                            metric='minkowski',
                            p=2,         # p=2 is equivalent to euclidian distance
                            metric_params=None,
                            n_jobs=1,
                            n_neighbors=3,
                            weights='uniform')

knn.fit(train_data, train_labels)
test_data_predicted = knn.predict(test_data)
print(accuracy_score(test_data_predicted, test_labels))
```

```
1.0
```

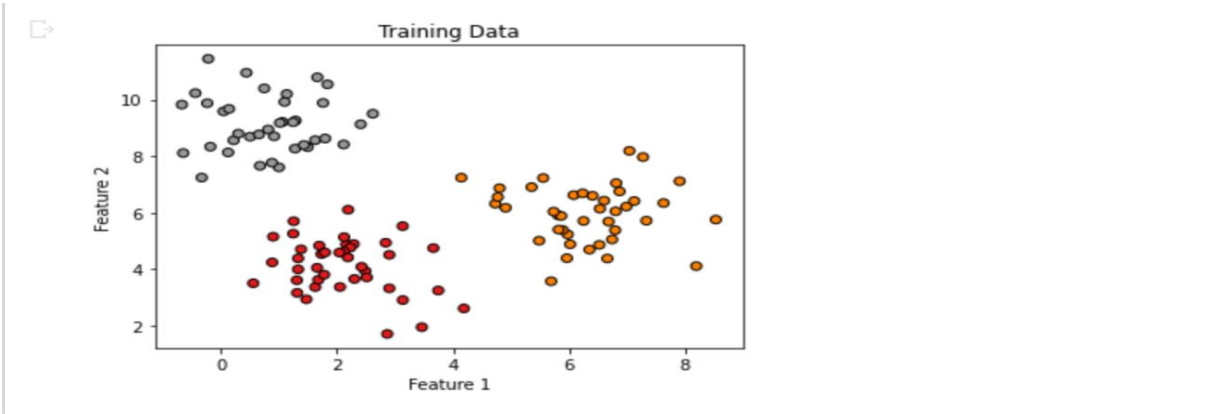
This code creates another instance of the KNeighborsClassifier class, called knn2, with some specific parameters set.

The algorithm parameter is set to 'auto', which means that the algorithm will automatically select the most appropriate algorithm based on the characteristics of the data.

The leaf_size parameter sets the size of the leaf nodes in the tree-based algorithms, which affects the speed and memory usage of the algorithm. The metric parameter sets the distance metric used to calculate distances between samples. In this case, 'minkowski' is used, which is a generalization of the Euclidean distance metric that allows for different values of the parameter p. The p parameter is set to 2, which is equivalent to the Euclidean distance metric. The n_neighbors parameter sets the number of neighbors to consider when making predictions. In this case, n_neighbors is set to 3, which means that the label of a test sample will be predicted based on the majority class of its three closest neighbors. The weights parameter sets the weighting strategy used when making predictions. In this case, 'uniform' is used, which means that all neighbors are given equal weight in the prediction. Other options include 'distance', which weights neighbors by the inverse of their distance to the test sample. The knn2 object can now be trained on the training data and labels, and used to make predictions on new, unseen data.

```
# Plot the data
plt.scatter(train_data[:, 0], train_data[:, 1], c=train_labels, cmap=plt.cm.Set1,
            edgecolor='k')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Training Data')
plt.show()
```

This code creates a scatter plot of the training data using matplotlib.



```
# Plot the decision boundary
h = .02  # step size in the mesh
x_min, x_max = train_data[:, 0].min() - 1, train_data[:, 0].max() + 1
y_min, y_max = train_data[:, 1].min() - 1, train_data[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Set1, shading='auto')

# Plot the training points
plt.scatter(train_data[:, 0], train_data[:, 1], c=train_labels, cmap=plt.cm.Set1,
            edgecolor='k')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Decision Boundary of Training Data')
plt.show()
```

This code plots the decision boundary of the k-nearest neighbors (KNN) classifier trained on the training data.

The scatter function is used to plot the training data, as in the previous code block. The resulting plot shows the decision boundary of the KNN classifier, with regions belonging to different classes (determined by the classifier) colored differently.

Summary:

The project imports several Python libraries and creates a synthetic dataset using the make_blobs function from scikit-learn. It then splits the dataset into a training set and a test set. Later splits the dataset into a training set and a test set and then trains a k-nearest neighbors (KNN) classifier on the training set using the KNeighborsClassifier. The default parameters are used for the classifier. And then trains a second KNN classifier on the training set with some specific parameters set. The results shown in a scatter plot of the training data using matplotlib, with points belonging to different classes colored differently and plots the decision boundary of the KNN classifier on the training data. The regions belonging to different classes are colored differently, and the decision boundary separates these regions.

Overall, this project demonstrates how to train a KNN classifier on a synthetic dataset and visualize the results using matplotlib. It also shows how to change some of the parameters of the KNN classifier to achieve different performance characteristics.