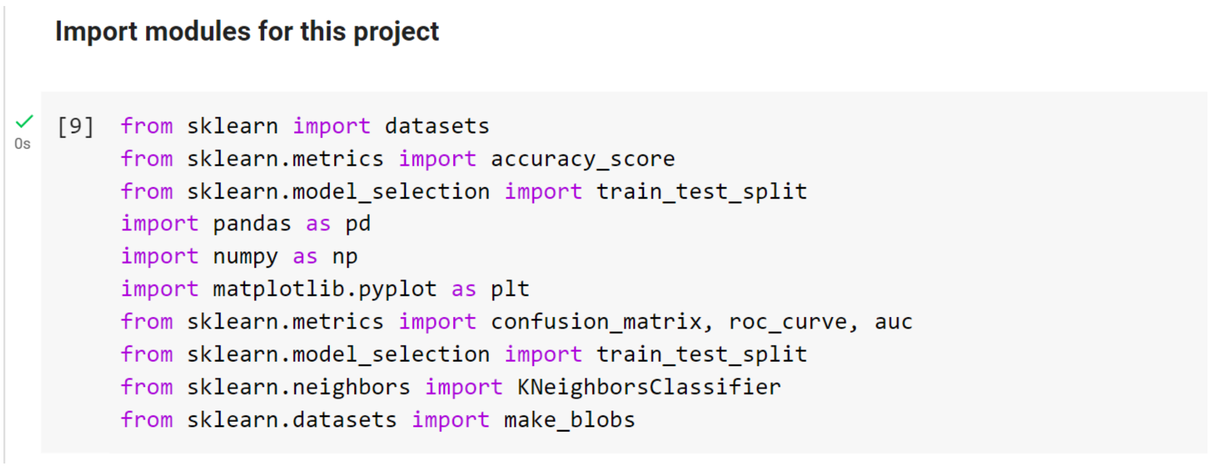
Introduction:

The Project shows how to create and test a k-nearest neighbors (KNN) classifier using Python and scikit-learn, a well-known machine learning toolkit. A common machine learning approach for classification and regression issues is KNN. The make blobs function of scikit-learn is used to create a synthetic dataset at the beginning of the code, which results in a dataset of points clustered into three separate categories. Using the train test split function of scikit-learn, the data is then divided into a training set and a test set. The KNeighborsClassifier class is then used to train the KNN classifier on the training set of data, and the training set is subsequently used to assess the classifier's performance. The code also explains how to see the classifier's decision boundary and build a scatter plot of the data. 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.

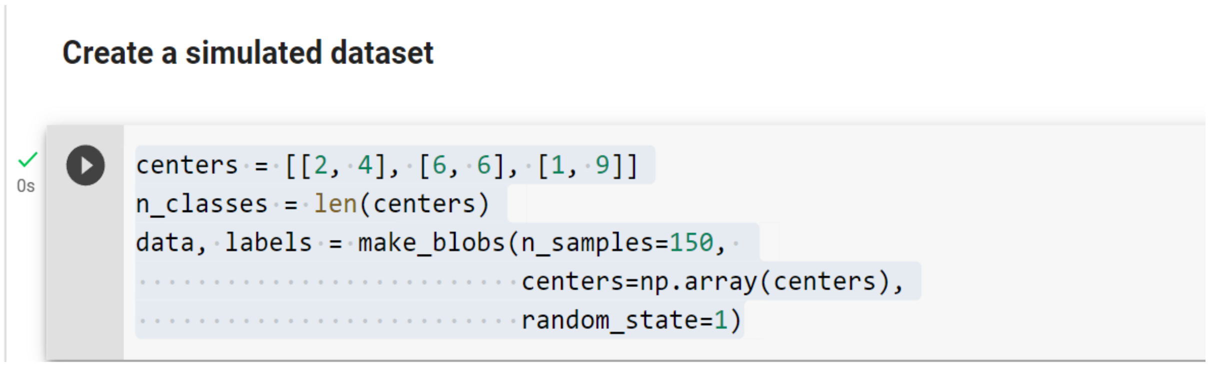


Import modules for this project

```
[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
```

For this project, the relevant modules have been imported. Data processing, modeling, and evaluation components are all included in the Python machine learning package known as sklearn. Other libraries for data analysis and manipulation include pandas and numpy. A visualization library for plotting and graphing data is called matplotlib. The simulated dataset is produced using the make_blobs function.

Data:

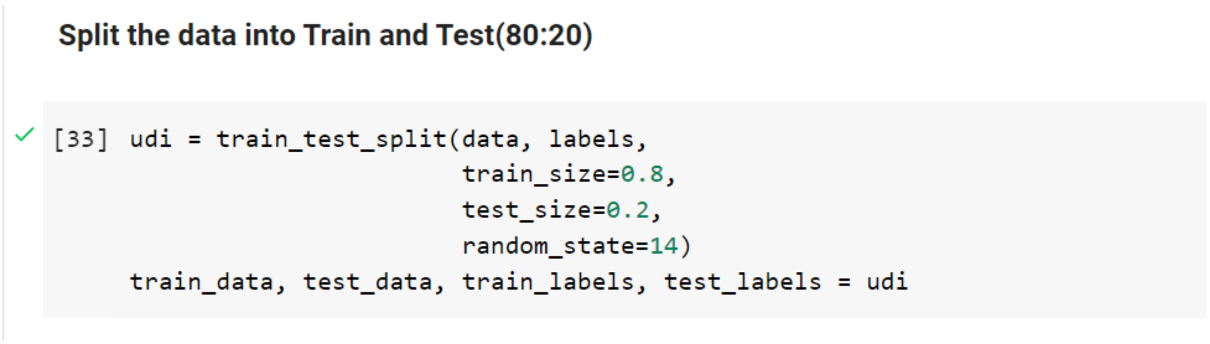


Create a simulated dataset

```
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 code snippet above generates a fictitious dataset with 150 records. Three clusters, each with a distinct center, are created as a result of how the dataset is constructed. The centers list, which contains the two-dimensional points [2, 4], [6, 6], and [1, 9], defines the centers of the three clusters. The number of clusters in the fictitious dataset, 3, is the value set for the n classes variable.

By randomly selecting points from a Gaussian distribution centered at each of the three cluster centers, the make blobs method creates the dataset. Each cluster's standard deviation is by default set to 1.0. Two arrays, data and labels, containing the two-dimensional feature vectors for each sample in the final dataset, 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.



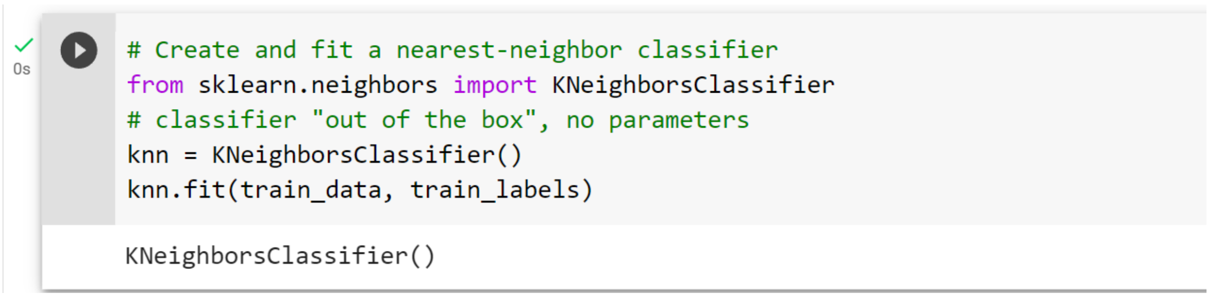
Split the data into Train and Test(80:20)

```
✓ [33] udi = train_test_split(data, labels,
                             train_size=0.8,
                             test_size=0.2,
                             random_state=14)
       train_data, test_data, train_labels, test_labels = udi
```

The train test split function from the sklearn.model selection module is used in the aforementioned code snippet to divide the synthetic dataset into training and testing groups. The dataset (data) and associated labels (labels) are passed into the train test split function, which divides them into different subsets for training and testing. The percentage of the dataset to be used for training and testing, respectively, is specified by the train size and test size variables. In this instance, training uses 80% of the data, and testing uses 20% of the data.

The random seed is set by the random state argument, ensuring that the same split is produced each time the function is executed. The ability to receive the same results each time the code is performed is crucial for reproducibility.

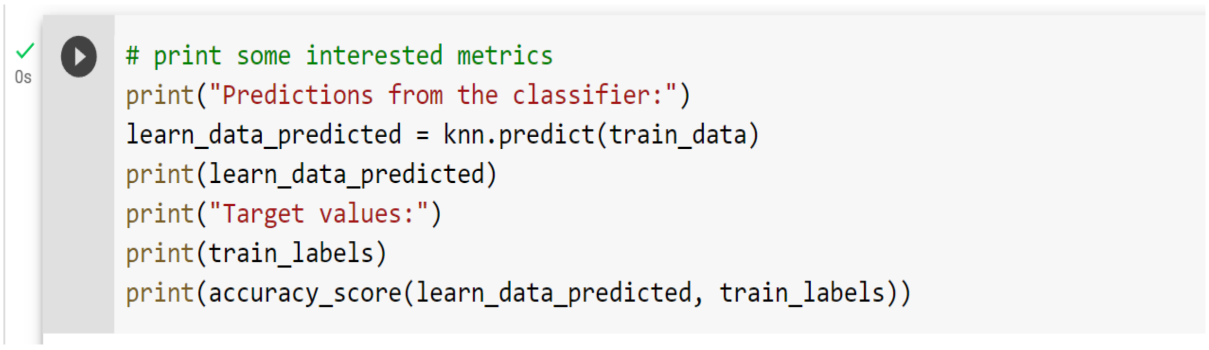
A tuple consisting of the four arrays train data, test data, train labels, and test labels is what the train test split function produces. The feature vectors for the training and testing subsets are respectively contained in the train data and test data arrays. The appropriate labels for the training and testing subsets are contained in the train labels and test labels arrays, respectively. For use in later training and testing of a machine learning model, these arrays are each allocated to the variables train data, test data, train labels, and test labels, in that order.



```
✓
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()
```

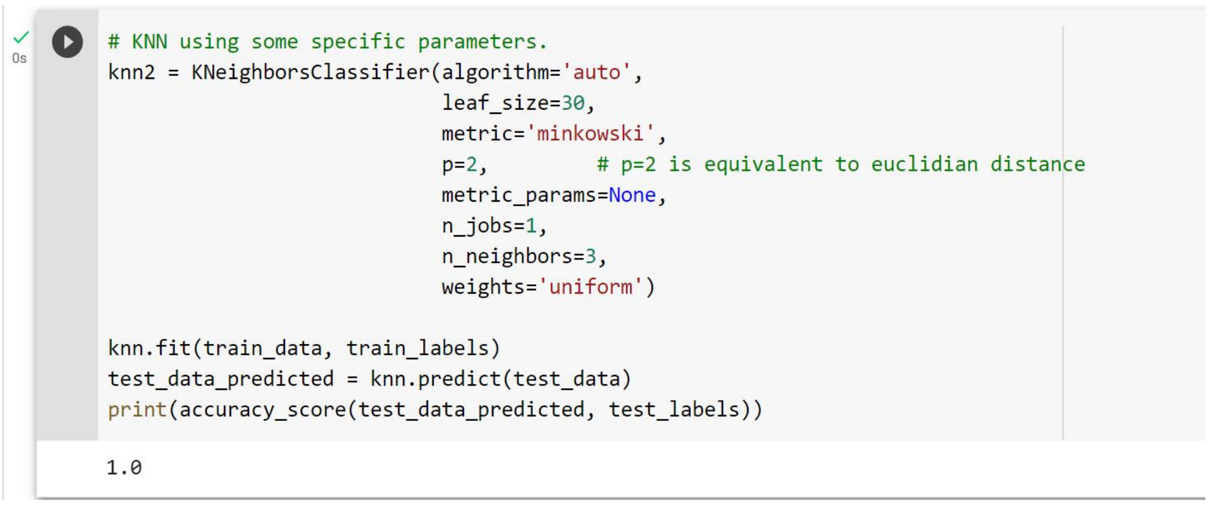
This code builds a nearest-neighbor classifier using the KNeighborsClassifier class from the sklearn.neighbors module. The k-nearest neighbors algorithm is a type of instance-based learning that uses a distance metric to identify the k-closest samples in the training data to a given test sample and then predicts the label of the test sample based on the majority class of its k nearest neighbors. The k-nearest neighbors class is an implementation of this algorithm. With no parameters supplied, the knn object is initialized as an instance of the KNeighborsClassifier class. This implies that the settings are used using their default values, which include utilizing the Euclidean distance metric and k=5 to determine the distances between samples. Then the knn object's fit method is invoked 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 trained nearest-neighbor classifier's evaluation metrics are printed out by the aforementioned code snippet. The learn data predicted variable contains the predictions for the training data that were produced using the predict method of the knn object. The code then prints the true labels (train labels) and the predicted labels for the training data (learn data predicted). This enables us to observe the classifier's performance on the training set. The accuracy score functio is used by the code to calculate the classifier's accuracy using the training data. The accuracy score function takes the true and predicted labels as inputs and outputs the percentage of samples for which the true and predicted labels are the same.

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
```

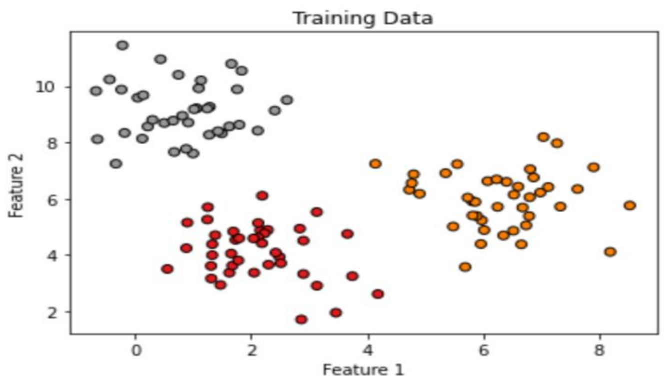
With a few precise parameters supplied, this code produces a second instance of the KNeighborsClassifier class, designated as knn2.

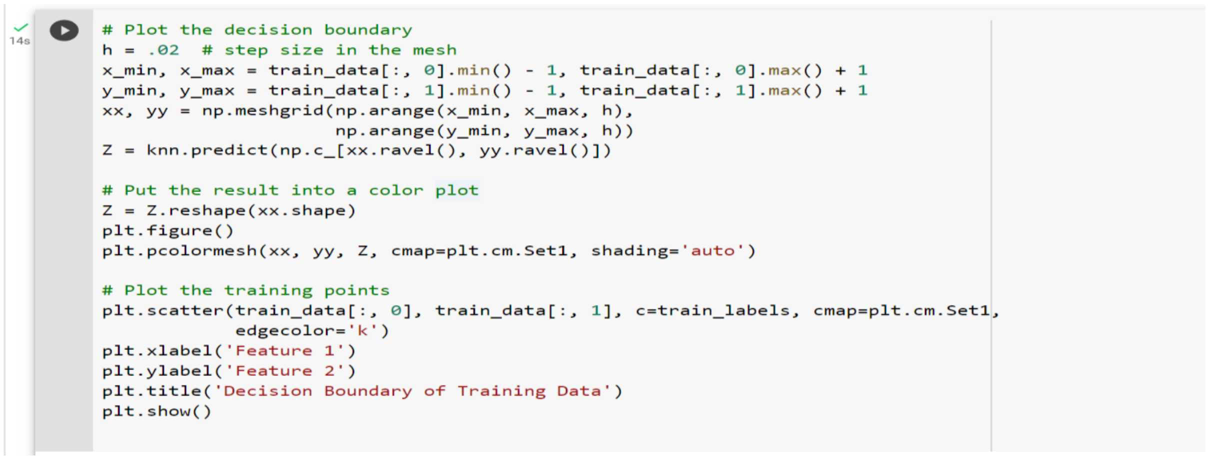
When the algorithm parameter is set to "auto," the algorithm will automatically choose the method that best fits the data's properties.

The leaf size parameter controls the size of the leaf nodes in tree-based algorithms, which has an impact on the algorithm's performance and memory requirements. The distance metric used to determine the separation between samples is set by the metric parameter. The "minkowski" distance metric, a modification of the Euclidean distance metric that accepts a range of values for the parameter p, is employed in this situation. The Euclidean distance metric is represented by the value 2 of the p parameter. The number of neighbors to take into account while making predictions is determined by the n neighbors option. The label of a test sample will be predicted based on the majority class of its three nearest neighbors in this instance because n neighbors is set to 3, which indicates that. 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.





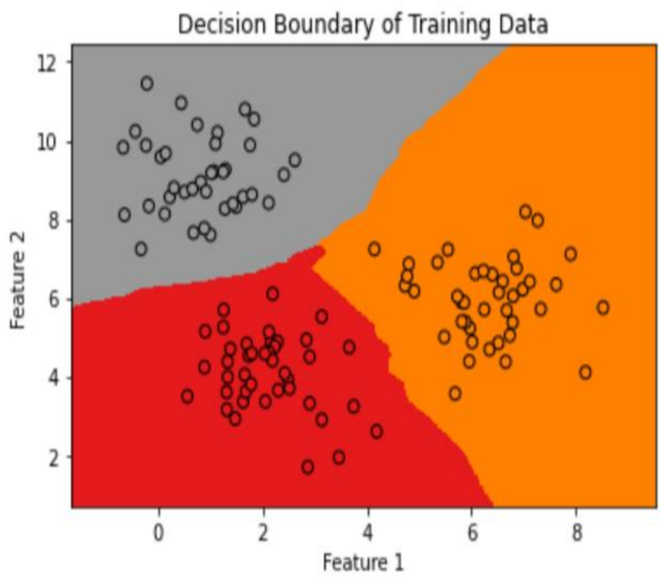
```
# Plot the decision boundary
h = .02  # step size in the mesh
x_min, x_max = train_data[:, 0].min() - 1, train_data[:, 0].max() + 1
y_min, y_max = train_data[:, 1].min() - 1, train_data[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
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 training data are plotted using the scatter function, just like in the preceding code block. The resulting figure displays the KNN classifier's decision border, with regions that belong to various classes (as decided by the classifier) being colored differently.



Summary:

The project imports a number of Python packages and uses scikit-make blobs learn's method to generate a fictitious dataset. After that, a training set and a test set are created from the dataset. Thereafter, the dataset is divided into a training set and a test set. The training set is then used to train a k-nearest neighbors (KNN) classifier using the KNeighborsClassifier. The classifier operates using its default settings. then after that uses the training set to build a second KNN classifier with a set of particular parameters. The outcomes are displayed in a scatter plot of the training data created with matplotlib, where points from different classes are colored differently and the KNN classifier's decision boundary is plotted on top of the training data. Several regions in each class are colored differently, and the decision boundary separates these regions.

Overall, this project shows how to use matplotlib to visualize the output after training a KNN classifier on a synthetic dataset. It also demonstrates how to alter a few KNN classifier parameters to obtain various performance traits.