Scikit-Learn

Created in 2007 by David Cournapeau as part of a Google Summer of Code project, scikit-learn is an open source Python library made to facilitate the process of building models based on built-in ML and statistical algorithms, without the need for hardcoding. The main reasons for its popular use are its complete documentation, its easy-to-use API, and the many collaborators who work every day to improve the library.

Note

You can find the documentation for scikit-learn at http://scikit-learn.org.

Scikit-learn is mainly used to model data, and not as much to manipulate or summarize data. It offers its users an easy-to-use, uniform API to apply different models with little learning effort, and no real knowledge of the math behind it is required.

In these exercises, we will explore the main steps for working on a supervised machine learning problem. First, this note explains the different sets in which data needs to be split for training, validating, and testing your model. Next, the most common evaluation metrics will be explained.

Split Ratio

It is important to clarify the split ratio in which data needs to be divided. Although there is no exact science for calculating the split ratio, there are a couple of things to consider when doing so:

Size of the dataset: Previously, when data was not easily available, datasets contained between 100 and 100,000 instances, and the conventionally accepted split ratio was 60/20/20% for the training, validation, and testing sets, respectively.

With software and hardware improving every day, researchers can put together datasets that contain over a million instances. This capacity to gather huge amounts of data allows the split ratio to be 98/1/1%, respectively. This is mainly because the larger the dataset, the more data can be used for training a model, without compromising the amount of data left for the validation and testing sets.

The following diagram displays the proportional partition of the dataset into three subsets. It is important to highlight that the training set must be larger than the other two, as it is the one to be used for training the model. Additionally, it is possible to observe that both the training and validation sets have an effect on the model, while the testing set is mainly used to validate the actual performance of the model with unseen data. Considering this, the training and validation sets must come from the same distribution:

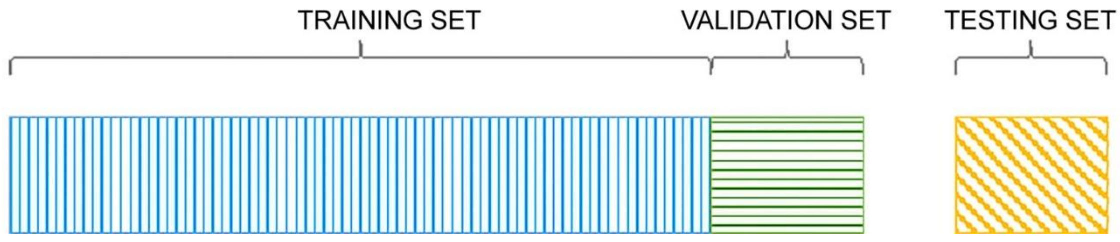


Figure 1: Visualization of the split ratio

Exercise 1: Performing a Data Partition on a Sample Dataset

In this exercise, we will be performing a data partition on the `wine` dataset using the split ratio method. The partition in this exercise will be done using the three-splits approach. Follow these steps to complete this exercise:

Note

For the exercises and activities within this chapter, you will need to have Python 3.7, NumPy, Jupyter, Pandas, and scikit-learn installed on your system.

1. Open a Jupyter Notebook to implement this exercise. Import the required elements, as well as the **load_wine** function from scikit-learn's **datasets** package:
   ```
   from sklearn.datasets import load_wine
   import pandas as pd
   from sklearn.model_selection import
   train_test_split
   ```
   The first line imports the function that will be used to load the dataset from scikit-learn. Next, **pandas** library is imported. Finally, the **train_test_split** function is imported, which will be in charge of partitioning the dataset. The function partitions the data into two subsets (a train and a test set). As the objective of this exercise is to partition data into three subsets, the function will be used twice to achieve the desired result.
2. Load the **wine** toy dataset and store it in a variable named **data**. Use the following code snippet to do so:
   ```
   data = load_wine()
   ```
   The **load_wine** function loads the toy dataset provided by scikit-learn.

   Note
   To check the characteristics of the dataset, visit the following link: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_wine.html.

   The output from the preceding function is a dictionary-like object, which separates the features (callable as data) from the target (callable as target) into two attributes.

3. Convert each attribute (data and target) into a Pandas DataFrame to facilitate data manipulation. Print the shape of both DataFrames:

```
X = pd.DataFrame(data.data)
Y = pd.DataFrame(data.target)
print(X.shape,Y.shape)
```

The output from the **`print`** function should be as follows:

```
(178, 13) (178, 1)
```

Here, the values in the first parenthesis represent the shape of DataFrame **`X`** (known as the features matrix), while the values in the second parenthesis refer to the shape of DataFrame **`Y`** (known as the target matrix).

4. Perform your first split of the data using the **`train_test_split`** function. Use the following code snippet to do so:

```
X, X_test, Y, Y_test = train_test_split(X, Y,
test_size = 0.2)
```

The inputs of the **`train_test_split`** function are the two matrices **`(X,Y)`** and the size of the test set, as a value between 0 and 1, which represents the proportion.

Note

Considering that we are dealing with a small dataset, as per the explanation in the Split Ratio section, we're using a split ratio of 60/20/20%. Remember that for larger datasets, the split ratio usually changes to 98/1/1%.

The outputs of the preceding function are four matrices: **`X`** divided into two subsets (train and test) and **`Y`** divided into two corresponding subsets:

```
print(X.shape, X_test.shape, Y.shape, Y_test.shape)
```

By printing the shape of all four matrices, as per the preceding code snippet, it is possible to confirm that the size of the test subset (both **`X`** and **`Y`**) is 20% of the total size of the original dataset (150 * 0.2 = 35.6) rounded to an integer, while the size of the train set is the remaining 80%:

```
(142, 13) (36, 13) (142, 1) (36, 1)
```

5. To create a validation set (dev set), we will use the **`train_test_split`** function to divide the train sets we

obtained in the previous step. However, to obtain a dev set that's the same shape as the test set, it is necessary to calculate the proportion of the size of the test set over the size of the train set before creating a validation set. This value will be used as the **`test_size`** for the next step:

```
dev_size = 36/142
print(dev_size)
```

Here, **`36`** is the size of the test set we created in the previous step, while **`142`** is the size of the train set that will be further split. The result from this operation is around **`0.25`**, which can be verified using the **`print`** function.

6. Use the **`train_test_split`** function to divide the train set into two subsets (train and dev sets). Use the result from the operation in the previous step as the **`test_size`**:

```
X_train, X_dev, Y_train, Y_dev = 
train_test_split(X, Y, \
                                 test_size = 
dev_size)
print(X_train.shape, Y_train.shape, X_dev.shape, \
      Y_dev.shape, X_test.shape, Y_test.shape)
```

The output of the **`print`** function is as follows:

```
(106, 13) (106, 1) (36, 13) (36, 1) (36, 13) (36, 
1)
```

You have successfully split the dataset into three subsets to develop efficient machine learning projects. Feel free to test different split ratios.

In conclusion, the split ratio to partition data is not fixed and should be decided by taking into account the amount of data available, the type of algorithm to be used, and the distribution of the data.

Cross-Validation

Cross-validation is also a procedure that's used to partition data by resampling the data that's used to train and validate the model. It consists of a parameter, *K*, that represents the number of groups that the dataset will be divided into.

Due to this, the procedure is also referred to as K-fold cross-validation, where *K* is usually replaced by a number of your choice. For instance, a model that's created using a 10-fold cross-validation procedure signifies a model where data is divided into 10 subgroups. The procedure of cross-validation is illustrated in the following diagram:

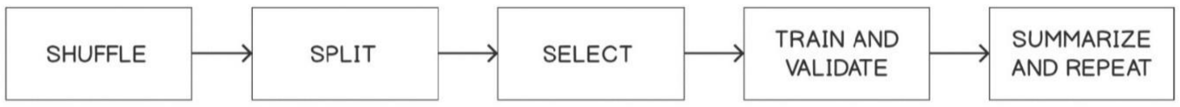


Figure 2: Cross-validation procedure

There is no exact science to choosing the value for *K*, but it is important to consider that lower values for *K* tend to decrease variance and increase bias, while higher *K* values result in the opposite behaviour. Also, the lower *K* is, the less expensive the processes, which results in faster running times.

Exercise 2: Using Cross-Validation to Partition the Train Set into a Training and a Validation Set

In this exercise, we will be performing a data partition on the `wine` dataset using the cross-validation method. Follow these steps to complete this exercise:

1. Import all the required elements:
```
from sklearn.datasets import load_wine
import pandas as pd
from sklearn.model_selection import
train_test_split
from sklearn.model_selection import KFold
```

The last line in the preceding code imports the **`KFold`** class from scikit-learn, which will be used to partition the dataset.

2. Load the **`wine`** dataset as per the previous exercise and create the Pandas DataFrames containing the features and target matrices:

```
data = load_wine()
X = pd.DataFrame(data.data)
Y = pd.DataFrame(data.target)
```

3. Split the data into training and testing sets using the **`train_test_split`** function, which you learned about in the previous exercise, using a **`test_size`** of 0.10:

```
X, X_test, Y, Y_test = train_test_split(X, Y, \
                                        test_size =
0.10)
```

4. Instantiate the **`KFold`** class with a 10-fold configuration:

```
kf = KFold(n_splits = 10)
```

Note

Feel free to experiment with the values of **`K`** *to see how the output shapes of this exercise vary.*

5. Apply the **`split`** method to the data in **`X`**. This method will output the index of the instances to be used as training and validation sets. This method creates 10 different split configurations. Save the output in a variable named **`splits`**:

```
splits = kf.split(X)
```

Note that it is not necessary to run the **`split`** method on the data in **`Y`**, as the method only saves the index numbers, which will be the same for **`X`** and **`Y`**. The actual splitting is handled next.

6. Perform a **`for`** loop that will go through the different split configurations. In the loop body, create the variables that will hold the data for the training and validation sets. Use the following code snippet to do so:

```
for train_index, test_index in splits:
    X_train, X_dev = X.iloc[train_index,:], \
                     X.iloc[test_index,:]
    Y_train, Y_dev = Y.iloc[train_index,:], \
                     Y.iloc[test_index,:]
```

The **for** loop goes through **K** number of configurations. In the body of the loop, the data is split using the index numbers:

```
print(X_train.shape, Y_train.shape, X_dev.shape, \
      Y_dev.shape, X_test.shape, Y_test.shape)
```

By printing the shape of all the subsets, as per the preceding snippet, the output is as follows:

```
(144, 13) (144, 1) (16, 13) (16, 1) (18, 13) (18,
1)
```

Note

The code to train and evaluate the model should be written inside the loop body, given that the objective of the cross-validation procedure is to train and validate the model using the different split configurations.

You have successfully performed a cross-validation split on a sample dataset.

In conclusion, cross-validation is a procedure that's used to shuffle and split the data into training and validation sets so that the process of training and validating is done each time on a different set of data, thus achieving a model with low bias.

Exercise 3 (ACTIVITY, try coding based on first two exercises): Data Partitioning on a Handwritten Digit Dataset

Your company specializes in recognizing handwritten characters. It wants to improve the recognition of digits, which is why they have gathered a dataset of 1,797 handwritten digits from 0 to 9. The images have already been converted into their numeric representation, and so they have provided you with the dataset to split it into training/validation/testing sets. You can choose to either perform conventional splitting or cross-validation. Follow these steps to complete this activity:

1. Import all the required elements to split a dataset, as well as the `load_digits` function from scikit-learn to load the `digits` dataset.
2. Load the `digits` dataset and create Pandas DataFrames containing the features and target matrices.
3. Take the conventional split approach, using a split ratio of 60/20/20%.
4. Using the same DataFrames, perform a 10-fold cross-validation split.

Evaluation Metrics

Model evaluation is indispensable for creating effective models that not only perform well on the data that was used to train the model but also on unseen data. The task of evaluating the model is especially easy when dealing with supervised learning problems, where there is a ground truth that can be compared against the prediction of the model.

Evaluation Metrics for Classification Tasks

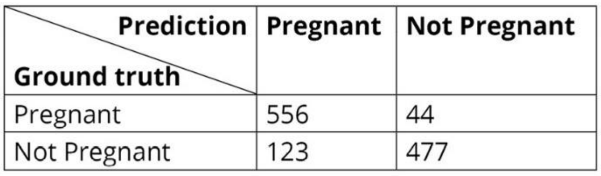
Now, we will take a look at the different performance metrics.

Confusion Matrix

The **confusion matrix** is a table that contains the performance of the model, and is described as follows:

- The columns represent the instances that belong to a predicted class.
- The rows refer to the instances that actually belong to that class (ground truth).

The configuration that confusion matrices present allows the user to quickly spot the areas in which the model is having greater difficulty. Consider the following table:



Prediction / Ground truth	Pregnant	Not Pregnant
Pregnant	556	44
Not Pregnant	123	477

Figure 3: A confusion matrix of a classifier that predicts whether a woman is pregnant

The following can be observed from the preceding table:

- By summing up the values in the first row, it is possible to know that there are 600 observations of pregnant women. However, from those 600 observations, the model predicted 556 as pregnant, and 44 as non-pregnant. Hence, the model's ability to predict that a woman is pregnant has a correctness level of 92.6%.
- Regarding the second row, there are also 600 observations of non-pregnant women. Out of those 600, the model predicted that 123 of them were pregnant, and 477 were non-pregnant. The model successfully predicted non-pregnant women 79.5% of the time.

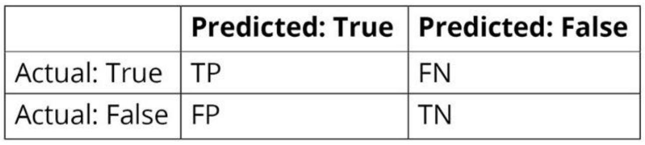
Based on these statements, it is possible to conclude that the model performs at its worst when classifying observations that are not pregnant.

Considering that the rows in a confusion matrix refer to the occurrence or non-occurrence of an event, and the columns refer to the model's predictions, the values in the confusion matrix can be explained as follows:

- **True positives (TP)**: Refers to the instances that the model correctly classified the event as positive—for example, the instances correctly classified as pregnant.
- **False positives (FP)**: Refers to the instances that the model incorrectly classified the event as positive—for example, the non-pregnant instances that were incorrectly classified as pregnant.

- **True negatives (TN)**: Represents the instances that the model correctly classified the event as negative—for example, the instances correctly classified as non-pregnant.
- **False negatives (FN)**: Refers to the instances that the model incorrectly classified the event as negative—for example, the pregnant instances that were incorrectly predicted as non-pregnant.

The values in the confusion matrix can be demonstrated as follows:



	Predicted: True	**Predicted: False**
Actual: True	TP	FN
Actual: False	FP	TN

Figure 4: A table showing confusion matrix values

Accuracy

Accuracy, as explained previously, measures the model's ability to correctly classify all instances. Although this is considered to be one of the simplest ways of measuring performance, it may not always be a useful metric when the objective of the study is to minimize/maximize the occurrence of one class independently of its performance on other classes.

The accuracy level of the confusion matrix from *Figure 3* is measured as follows:

$$Accuracy = \frac{(TP + TN)}{m} = 0.8608 \approx 86\%$$

Figure 5: An equation showing the calculation for accuracy

Here, *m* is the total number of instances.

The `86%` accuracy refers to the overall performance of the model in classifying both class labels.

Precision

This metric measures the model's ability to correctly classify positive labels (the label that represents the occurrence of the event) by comparing it with the total number of instances predicted as positive. This is represented by the ratio between the *true positives* and the sum of the *true positives* and *false positives*, as shown in the following equation:

$$Precision = \frac{TP}{TP + FP}$$

Figure 6: An equation showing the calculation for precision

The precision metric is only applicable to binary classification tasks, where there are only two class labels (for instance, true or false). It can also be applied to multiclass tasks considering that the classes are converted into two (for instance, predicting whether a handwritten number is a 6 or any other number), where one of the classes refers to the instances that have a condition while the other refers to those that do not.

For the example in *Figure 3*, the precision of the model is equal to 81.8%.

Recall

The recall metric measures the number of correctly predicted positive labels against all positive labels. This is represented by the ratio between *true positives* and the sum of *true positives* and *false negatives*:

$$Recall = \frac{TP}{TP + FN}$$

Figure 7: An equation showing the calculation for recall

Again, this measure should be applied to two class labels. The value of recall for the example in *Figure 3* is 92.6%, which, when compared to the other two metrics, represents the highest performance of the model. The decision to choose one metric or the other will depend on the purpose of the study, which will be explained in more detail later.

Exercise 4: Calculating Different Evaluation Metrics on a Classification Task

In this exercise, we will be using the breast cancer toy dataset to calculate the evaluation metrics using the scikit-learn library. Follow these steps to complete this exercise:

1. Import all the required elements:
   ```
   from sklearn.datasets import load_breast_cancer
   import pandas as pd
   from sklearn.model_selection import
   train_test_split
   from sklearn import tree
   from sklearn.metrics import confusion_matrix
   from sklearn.metrics import accuracy_score
   from sklearn.metrics import precision_score
   from sklearn.metrics import recall_score
   ```
   The fourth line imports the **`tree`** module from scikit-learn, which will be used to train a decision tree model on the training data in this exercise. The lines of code below that will import the different evaluation metrics that will be calculated during this exercise.
2. The breast cancer toy dataset contains the final diagnosis (malignant or benign) of the analysis of masses found in the breasts of 569 women. Load the dataset and create features and target Pandas DataFrames, as follows:
   ```
   data = load_breast_cancer()
   X = pd.DataFrame(data.data)
   Y = pd.DataFrame(data.target)
   ```
3. Split the dataset using the conventional split approach:
   ```
   X_train, X_test, \
   Y_train, Y_test = train_test_split(X,Y, test_size =
   0.1, \
   ```

```
                                                    random_state =
0)
```

Note that the dataset is divided into two subsets (train and test sets) because the purpose of this exercise is to learn how to calculate the evaluation metrics using the scikit-learn package.

Note

The **`random_state`** *parameter is used to set a seed that will ensure the same results every time you run the code. This guarantees that you will get the same results as the ones reflected in this exercise. Different numbers can be used as the seed; however, use the same number as suggested in the exercises and activities of this chapter to get the same results as the ones shown here.*

4. First, instantiate the **`DecisionTreeClassifier`** class from scikit-learn's **`tree`** module. Next, train a decision tree on the train set. Finally, use the model to predict the class label on the test set. Use the following code to do this:

```
model = tree.DecisionTreeClassifier(random_state =
0)
model = model.fit(X_train, Y_train)
Y_pred = model.predict(X_test)
```

First, the model is instantiated using a **`random_state`** to set a seed. Then, the **`fit`** method is used to train the model using the data from the train sets (both **`X`** and **`Y`**). Finally, the **`predict`** method is used to trigger the predictions on the data in the test set (only **`X`**). The data from **`Y_test`** will be used to compare the predictions with the ground truth.

5. Use scikit-learn to construct a confusion matrix, as follows:

```
confusion_matrix(Y_test, Y_pred)
```

The result is as follows, where the ground truth is measured against the prediction:

```
array([[21, 1],
       [6, 29]])
```

6. Calculate the accuracy, precision, and recall of the model by comparing **`Y_test`** and **`Y_pred`**:

```
accuracy = accuracy_score(Y_test, Y_pred)
print("accuracy:", accuracy)
precision = precision_score(Y_test, Y_pred)
print("precision:", precision)
recall = recall_score(Y_test, Y_pred)
print("recall:", recall)
```

The results are displayed as follows:

```
accuracy: 0.8771
precision: 0.9666
recall: 0.8285
```

Given that the positive labels are those where the mass is malignant, it can be concluded that the instances that the model predicts as malignant have a high probability (96.6%) of being malignant, but for the instances predicted as benign, the model has a 17.15% (100%–82.85%) probability of being wrong.

You have successfully calculated evaluation metrics on a classification task.

Choosing an Evaluation Metric

There are several metrics that can be used to measure the performance of a model on classification tasks, and selecting the right one is key to building a model that performs exceptionally well for the purpose of the study.

Evaluation Metrics for Regression Tasks

Considering that regression tasks are those where the final output is continuous, without a fixed number of output labels, the comparison between the ground truth and the prediction is based on the proximity of the values rather than on them having exactly the same values. For instance, when predicting house prices, a model that predicts a value of USD 299,846 for a house valued at USD 300,000 can be considered to be a good model.

The two metrics most commonly used for evaluating the accuracy of continuous variables are the **Mean Absolute Error** (**MAE**) and the **Root Mean Squared Error** (**RMSE**), which are explained here:

- **Mean Absolute Error**: This metric measures the average absolute difference between a prediction and the ground truth, without taking into account the direction of the error. The formula to calculate the MAE is as follows:

$$MAE = \frac{1}{m} * \sum_{i=1}^{m} |y_i - \hat{y}_i|$$

Figure 8: An equation showing the calculation of MAE

Here, *m* refers to the total number of instances, *y* is the ground truth, and *ŷ* is the predicted value.

- **Root Mean Squared Error**: This is a quadratic metric that also measures the average magnitude of error between the ground truth and the prediction. As its name suggests, the RMSE is the square root of the average of the squared differences, as shown in the following formula:

$$RMSE = \sqrt{\frac{1}{m} * \sum_{i=1}^{m} (y_i - \hat{y}_i)^2}$$

Figure 9: An equation showing the calculation of RMSE

Both these metrics express the average error, in a range from 0 to infinity, where the lower the value, the better the performance of the model. The main difference between these two metrics is that the MAE assigns the same weight of importance to all errors, while the RMSE squares the error, assigning higher weights to larger errors.

Considering this, the RMSE metric is especially useful in cases where larger errors should be penalized, meaning that outliers are taken into account in the measurement of performance. For instance, the RMSE metric can be used when a value that is off by 4 is more than twice as bad as being off by 2. The MAE, on the other hand, is used when a value that is off by 4 is just twice as bad as a value off by 2.

Exercise 5: Calculating Evaluation Metrics on a Regression Task

In this exercise, we will be calculating evaluation metrics on a model that was trained using linear regression. We will use the **boston** toy dataset for this purpose. Follow these steps to complete this exercise:

1. Import all the required elements, as follows:
```
from sklearn.datasets import load_boston
import pandas as pd
from sklearn.model_selection import
train_test_split
from sklearn import linear_model
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
import numpy as np
```
The fourth line imports the **linear_model** module from scikit-learn, which will be used to train a linear regression model on the training dataset. The lines of code that follow import two performance metrics that will be evaluated in this exercise.

2. For this exercise, the **boston** toy dataset will be used. This dataset contains data about 506 house prices in Boston. Use the following code to load and split the dataset, the same as we did for the previous exercises:
```
data = load_boston()
X = pd.DataFrame(data.data)
Y = pd.DataFrame(data.target)
X_train, X_test, Y_train, Y_test =
train_test_split(X,Y, \
                                   test_size = 0.1,
random_state = 0)
```

3. Train a linear regression model on the train set. Then, use the model to predict the class label on the test set, as follows:

```
model = linear_model.LinearRegression()
model = model.fit(X_train, Y_train)
Y_pred = model.predict(X_test)
```

As a general explanation, the **`LinearRegression`** class from scikit-learn's **`linear_model`** module is instantiated first. Then, the **`fit`** method is used to train the model using the data from the train sets (both **`X`** and **`Y`**). Finally, the **`predict`** method is used to trigger the predictions on the data in the test set (only **`X`**). The data from **`Y_test`** will be used to compare the predictions to the ground truth.

4. Calculate both the MAE and RMSE metrics:

```
MAE = mean_absolute_error(Y_test, Y_pred)
print("MAE:", MAE)
RMSE = np.sqrt(mean_squared_error(Y_test, Y_pred))
print("RMSE:", RMSE)
```

The results are displayed as follows:

```
MAE: 3.9357
RMSE: 6.4594
```

Note

The scikit-learn library allows you to directly calculate the MSE. To calculate the RMSE, the square root of the value obtained from the **`mean_squared_error()`** *function is calculated. By using the square root, we ensure that the values from MAE and RMSE are comparable.*

From the results, it is possible to conclude that the model performs well on the test set, considering that both values are close to zero. Nevertheless, this also means that the performance can still be improved.

You have now successfully calculated evaluation metrics on a regression task that aimed to calculate the prices of houses based on their characteristics. In the next activity, we will calculate the performance of a classification model that was created to recognize handwritten characters.

Exercise 6 (ACTIVITY, try coding based on previous exercises): Evaluating the Performance of the Model Trained on a Handwritten Dataset

You continue to work on creating a model to recognize handwritten digits. The team has built a model and they want you to evaluate the performance of the model. In this activity, you will calculate different performance evaluation metrics on a trained model. Follow these steps to complete this activity:

1. Import all the required elements to load and split a dataset in order to train a model and evaluate the performance of the classification tasks.
2. Load the `digits` toy dataset from scikit-learn and create Pandas DataFrames containing the features and target matrices.
3. Split the data into training and testing sets. Use 20% as the size of the testing set.
4. Train a decision tree on the train set. Then, use the model to predict the class label on the test set.
   Note
   To train the decision tree, revisit Exercise 3.04, Calculating Different Evaluation Metrics on a Classification Task.
5. Use scikit-learn to construct a confusion matrix.
6. Calculate the accuracy of the model.
7. Calculate the precision and recall. Considering that both the precision and recall can only be calculated on binary classification problems, we'll assume that we are only interested in classifying instances as the number `6` or `any other number`.
   To be able to calculate the precision and recall, use the following code to convert `Y_test` and `Y_pred` into a one-hot vector. A one-hot vector consists of a vector that only contains zeros and ones. For this activity, the 0 represents the *number 6*, while the 1 represents `any other number`.

This converts the class labels (**`Y_test`** and **`Y_pred`**) into binary data, meaning that there are only two possible outcomes instead of 10 different ones.
Then, calculate the precision and recall using the new variables:

```
Y_test_2 = Y_test[:]
Y_test_2[Y_test_2 != 6] = 1
Y_test_2[Y_test_2 == 6] = 0
Y_pred_2 = Y_pred
Y_pred_2[Y_pred_2 != 6] = 1
Y_pred_2[Y_pred_2 == 6] = 0
```

You should obtain the following values as the output:

```
Accuracy = 84.72%
Precision = 98.41%
Recall = 98.10%
```