# Study Unit 5 Data Analytics in Python

## Learning Outcomes

By the end of this unit, you should be able to:

1. Design Python programmes for performing data analytics
2. Analyse data using appropriate tools for data mining

## Overview

In this unit, we will discuss the implementation of two analytics techniques in Python: k-means clustering and decision trees. All these functions, modules, and algorithms can be found in the scikit-learn library. The scikit-learn library is a machine learning library written for the Python programming language. It features various modules such as classification, clustering, regression, etc. We will first develop Python programs to prepare the DataFrame for the different requirements of these sub-packages. We will then learn how the two analytics techniques can be carried out by Python programs and how their results can be extracted and presented.

## Chapter 1 Introduction to Scikit-Learn

### 1.1 Installing and Importing Scikit-Learn

Lesson Recording - Introduction to Scikit-Learn

It has become very common to carry out tasks for data analytics, statistical modelling or machine learning in Python recently. And the trend is rising. In fact, many Python packages in these areas have also been developed. One of the most common libraries used for these purposes is scikit-learn, a free machine learning library written for Python. In this study unit, we will write our code with the scikit-learn functions in JupyterLab.

One reason that scikit-learn has become one of the most common machine learning libraries for programming is its broad applicability and functionality. It features various algorithms for classification, regression, clustering, etc. In machine learning, programs are constructed with parameters such that they can “learn” from newly fed data. That is, they can automatically adjust and improve their behaviour according to the new “knowledge”. Below is a table of the most common algorithms that are available in scikit-learn.

Table 5.1 Most Common Algorithms Available in scikit-learn

|  |  |
| --- | --- |
| **Supervised learning** | |
| Linear Models | Gaussian Processes |
| Discriminant Analysis | Cross decomposition |
| Kernel ridge regression | Decision Trees |
| Support Vector Machines | Isotonic regression |
| Nearest Neighbours | Neural network models (supervised) |
| **Unsupervised learning** | |
| Gaussian mixture models | Novelty and Outlier Detection |
| Clustering | Density Estimation |
| Covariance estimation | Neural network models (unsupervised) |

Beside machine learning algorithms, scikit-learn also provides modules for model selection, visualisation, data transformation as well as example datasets. The website <https://scikit-learn.org/stable/user_guide.html> contains many details of the library.

Same as NumPy, matplotlib and pandas, we can simply use pip, the package installer of Python, to download and install scikit-learn.

|  |
| --- |
| pip install scikit-learn |

After installing scikit-learn using pip, we can import it into our program.

|  |
| --- |
| import sklearn |

Note that it is sklearn and not scikit-learn that refers to the scikit-learn library in the Python programs. Nevertheless, since the library is extraordinary extensive, programmers usually do not import the entire library. Instead, the common practice is to load the required algorithm or only its “estimator” object. For instance, if linear regression models are required for the analytics task, we can import the estimator LinearRegression from the module linear\_model.

|  |
| --- |
| from sklearn.linear\_model import LinearRegression |

Since each module has its own estimators, functions, etc., it is important to refer to the official websites for the correct spelling, including the cases of the names. It is not unusual that we need to load couple of them for a single analytics task. It is therefore important to put sufficient comments in the program to explain the purpose and use of each imported module.

In this study unit, we will demonstrate two scikit-learn algorithms, k-means clustering and decision trees, to show how the library, and Python in general, can be applied in data analytics. But before we can apply these algorithms, we need to prepare the data according to the requirements of each of the algorithms. The preparation process will be discussed in the next section.

|  |
| --- |
| **Example (Adult Census Data):** In this study unit, we will construct programs to estimate the two mentioned machine learning models on the US Adult Census dataset which has been introduced in Study Unit 4. The dataset is a repository of 48,842 entries extracted from the 1994 US Census database to predict whether income would exceed $50,000 per year according to the 14 social-demographic attributes. Before we begin to develop the code using the scikit-learn algorithms, we need to import the corresponding packages or modules first.    Figure 5.1 Importing Modules and Functions from scikit-learn  In the first box, the packages introduced in the two previous study units, pandas, NumPy and matplotlib are imported. We will need them to manage DataFrames, to convert slices of DataFrames to multidimensional arrays, and to construct plots to illustrate and evaluate the model results. In the second box, we import modules from scikit-learn that we need for the pre-processing and transformation of the DataFrames for model constructions. For instance, with the train\_test\_split function, we can instruct Python to split arrays into random training and testing subsets for evaluating the estimator performance. Furthermore, the module metrics includes functions to compute metrics and distances for the evaluation of classification performance. The functions in the preprocessing module such as scaling, centring, normalisation, etc. are used to prepare DataFrames for the scikit-learn algorithms. In the last two boxes, we import the modules of k-means clustering (KMeans) and decision trees (tree). In addition to the KMeans module, we also import the PCA module from the decomposition sub-package for dimension reduction, which will be helpful to plot multivariate data as a two-dimensional chart. |

**Read**

Refer to the link below for more details on the installation of the scikit-learn package:

<https://scikit-learn.org/stable/install.html>

Refer to the link below for more details and examples on the cluster module of the scikit-learn package for applying K-Means clustering:

<https://scikit-learn.org/stable/modules/clustering.html>

Refer to the link below for more details and examples on the tree module of the scikit-learn package for constructing decision trees:

<https://scikit-learn.org/stable/modules/tree.html>

Refer to the link below for more details and examples on the train\_test\_split module of the scikit-learn package for splitting arrays into random training and testing subsets:

[https://scikit-learn.org/stable/modules/generated/sklearn.model\_selection.train\_test\_‌split.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)

Refer to the link below for more details and examples on the metric module of the scikit-learn package for metrics and distance computations:

<https://scikit-learn.org/stable/modules/classes.html?highlight=metrics#module-sklearn.metrics>

Refer to the link below for more details and examples on the preprocessing module of the scikit-learn package for data preparation techniques such as scaling, centring, normalisation, etc.:

[https://scikit-learn.org/stable/modules/classes.html?highlight=preprocessing#‌module-sklearn.preprocessing](https://scikit-learn.org/stable/modules/classes.html?highlight=preprocessing#module-sklearn.preprocessing)

### 1.2 Data Preparation for Analytics Algorithms

Lesson Recording - Data Preparation for Analytics Algorithms of scikit-learn (1/2)

Lesson Recording - Data Preparation for Analytics Algorithms of scikit-learn (2/2)

In Figure 5.1, various modules for data preparation have been imported. At the same time, we have also imported packages that we have already worked with in the previous study units: NumPy, pandas, matplotlib. In fact, the scikit-learn algorithms work hand-in-hand with these packages. In this section, we will need to combine NumPy, pandas and scikit-learn to prepare datasets to meet the requirements of every scikit-learn module.

#### 1.2.1 Missing Values

One of the first steps in data preparation is to check on and to deal with missing values in the dataset. In Chapter 4 of Study Unit 4, we have discussed how to specify, identify, and modify observations with missing values. To recall some details, we can define specific strings in the dataset as missing values during the reading process and, at the same time, we instruct Python to treat white strings as missing values if necessary.

|  |
| --- |
| DataFrame\_name = pd.read\_csv("csv\_file\_name.csv", na\_values = "na\_string", na\_filer = True/False) |

After creating a pandas DataFrame, we have to decide on the appropriate measure to deal with the missing values in it. One way is to remove those rows with missing values in any of the columns from the DataFrame completely.

|  |
| --- |
| DataFrame\_name.dropna(axis = 0, how = "any"/"all") |

Another possibility is to replace them by specific values. Here, we can choose to apply the replacement on missing values of the entire DataFrame or just one specific column.

|  |
| --- |
| DataFrame\_name.fillna(value = repl\_value)  DataFrame\_name["column\_label"].fillna(value = repl\_value) |

The advantage of replacing the missing values in all columns is certainly the convenience in creating the corresponding code. Nevertheless, it is not unusual that a DataFrame contains various types of variables. In this case, replacing all missing values by a single value may be undesirable or impossible. The replacement values should be chosen according to the characteristics and requirements of each variable.

|  |
| --- |
| **Example (Cont’d):** After studying the missing data in the US Adult Census dataset, which occur solely in the variables workclass, occupation, and native-country, it seems appropriate to remove all these rows from the DataFrame.    Figure 5.2 Removing Missing Data from DataFrame  After removing the missing data from census, the DataFrame contains 45,222 rows. That is, we had a total of 3,620 observations that contain missing values originally. |

#### 1.2.2 Reducing Number of Categories

If a DataFrame contains categorical variables, they must be treated differently in comparison to scale or interval variables. In data analytics, we usually convert them to dummy variables in the pre-processing stage. We will discuss the conversion process in detail in Chapter 1.2.5. Nevertheless, if the variable contains a large number of categories, the number of dummy variables will become large as well. As a result, the analytics algorithm will have to handle a large number of variables and the required computational effort in the fitting process could be significant. One possible solution here is to reduce the number of categories at the expense of information loss. It is therefore a task for data analysts to balance this trade-off carefully.

Basically, the process of category reduction is to put observations from similar categories into a new category. For instance, if the country names are categories of a categorical variable, we can group them by their continents, and if a categorical variable contains the models of a certain product, we can group them by their brands or their main features. The similarity of the categories is essential here for not losing too much information.

In terms of programming, all we need to do is to replace some category labels in a variable by the .replace() method of the pandas package.

|  |
| --- |
| DataFrame\_Name["column\_label"].replace(to\_replace, value) |

The parameter to\_replace can be a list or dictionary of category labels to be replaced by the list or dictionary of new labels that is assigned to the value parameter.

|  |
| --- |
| **Example (Cont’d):** In the US Adult Census dataset, there are three relevant categorical variables that we would like to reduce their number of categories: workclass, occupation, and education (native-country may have the most categories among all variables, but it is rather irrelevant). In the first step, we list out the categories of workclass.    Figure 5.3 Listing Out Unique Categories of a Variable  For instance, we can group all the government employees into one group and those in self-employment into another. To assign new labels to the old ones, we first create a dictionary in which the keys contain the original labels, and the values represent the new ones.    Figure 5.4 Creating a Dictionary with Old and New Category Labels  Actually, it is not required to construct this dictionary first. We could have also put the corresponding labels in the .replace() method directly. Although this step may seem redundant in the first sight, the advantage here is that we can refer to this dictionary to retrieve the original labels whenever the program requires. By applying the .keys() and values() methods to the dictionary, we can convert all the keys and values into their own list for the use in the .replace() method.  To maintain the possibility of using the original variable in the algorithms eventually, we save the variable with reduced categories as a new column and label it workclass\_new.    Figure 5.5 Reducing the Number of Categories for a Categorical Variable  The newly created variable will appear at the rightmost column of the DataFrame. For marital-status, we follow the same steps to create a new column named marital-status\_new with less categories.    Figure 5.6 Reducing the Number of Categories for a Categorical Variable  In Figure 5.6, we can see that the different marriage statuses have been merged to a general group named “Married”. Furthermore, the label of the category “Never-married” has been shortened to “Single” to favour the visualisation of the result output later.  To reduce the categories in education, we can transform educational-num instead since they are equivalent. Moreover, educational-num is an ordered numeric variable so that we can apply discretisation as described in the following section. |

**Read**

Refer to the link below for more details and examples on the .replace() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌replace.html>

#### 1.2.3 Discretisation

If a categorical variable has ordered numeric values as categories, we can discretise them into new bins by the cut() function of the pandas package. The cut() function has been introduced in Chapter 5.2 of Study Unit 4.

|  |
| --- |
| DataFrame\_name["column"] = pd.cut(x = array, bins, labels) |

For our purpose here to reduce the number of categories, it is sufficient to put the highest value of each category in the list assigned to bins. When applying the cut() function, we have to be aware that it only includes the rightmost edge in each bin and not the leftmost one. Hence, the list for bins should start with 0, or -1 in case 0 is one of the numeric values of the original categories.

|  |
| --- |
| **Example (Cont’d):** As mentioned in the previous section, the variables education and educational-num are equivalent. We can show this by the crosstab() function where the frequency of each category in education will be counted when cross-combining with a value in educational-num.    Figure 5.7 Cross Tabulation of Two Categorical Variables  From Figure 5.7, we can see that there are only counts in one category in education for each value in educational-num. In other words, every category in educational-num corresponds to exactly one category in education. Furthermore, we can also see that the education level increases with the increase of the value in educational-num. For instance, “pre-school” level is 1 in educational-num while education level “1st-4th” has value 2. The highest education level in this dataset is “Doctorate” and its value in educational-num is also the highest, namely 16. To gain a better overview of the categories in both variables, we can chain the methods .groupby().mean().sort\_values() as in Figure 5.8.    Figure 5.8 Numeric Labels of a Categorical Variable  In the first step, we select the only required columns education and educational-num for this output. Then we use the .groupby() method to group the variables by the categories in education. At the same time, the mean for each category in educational-num will be calculated. Since the values in educational-num are the same for all observations in the same education category, their mean must be equal to their category value in educational-num. In the final step, we sort the rows of the grouped table by the values in educational-num since it is the ordered version of all the educational levels.  Eventually, we can reduce the number of categories in education by assigning the values in educational-num to new bins.    Figure 5.9 Discretising an Ordered Numeric Categorical Variable  In education\_new, we summarise “pre-school” (1) and all the primary (2-3) levels to the category “Primary”. All the secondary levels, including those who could not complete the college degree (4-10) are assigned to “Secondary”. Both “Assoc-voc” (11) and “Assoc-acdm” (12) are now in the group of “Associate”. While “Bachelor” (13) remains “Bachelor”, every observation with higher education levels than that (14-16) is now grouped into the category “Postgraduate”.  Since 0 does not belong to the value in educational-num, we can include it in the list assigned to the parameter bins and take it as the leftmost edge of the first bin. |

**Read**

Refer to the link below for more details and examples on the crosstab() function of the pandas package:

<https://pandas.pydata.org/docs/reference/api/pandas.crosstab.html>

#### 1.2.4 Selecting and Renaming Variables

It is very common that a dataset contains variables that are not directly relevant to be included in the analytics algorithm. Some of them could be redundant in their meaning; some of them are the original version of a transformed variable. These variables should be removed from the DataFrame before using the data to run the scikit-learn estimator.

In Chapter 2 of Study Unit 4, we have learned how to select rows and columns from the pandas DataFrame using index, Boolean masks, and localisation. The attributes .iloc() and .loc() are crucial in this context. We can use the same procedures to select the necessary columns for the scikit-learn algorithm.

If a dataset is originated from an external source, the given variable names may not necessarily reflect the needs and ideas of the analyst. Sometimes, they can be lengthy and make the result output visually appalling. In pandas, the .rename() method is used to rename the variables in a DataFrame.

|  |
| --- |
| DataFrame\_name.rename(columns = {"oldvar": "newvar"}) |

The column labels to be renamed must be put as keys of a dictionary that will be assigned to the parameter columns in the .rename() method. The values of the dictionary will then be the new labels of the corresponding columns.

|  |
| --- |
| **Example (Cont’d):** After reducing the number of categories in workclass, marital-status and education, we have variables of both versions in our DataFrame. For the construction of the model, however, we only need the new ones. That is, we shall only select all categorical variables with the suffix “\_new”. Furthermore, fnlwgt is not relevant for our analyses and will be dropped. The variables occupation, relationship and native-country are not included in the models as well since they are correlated with the variables workclass, marital-status and race, respectively. After discretisation, educational-num also becomes redundant. Hence, it will be removed from the final selection.    Figure 5.10 Selecting Relevant Variables  We first create a list of the independent variables named X\_var and a list with the dependent variable named y\_var. They will then be concatenated before being selected from the census DataFrame by indexing. The resulting DataFrame is named DF\_model.  Since we have removed workclass, marital-status, and education from the DF\_model DataFrame, we can now rename workclass\_new, marital-status\_new, and education\_new back to the names of the original variables.    Figure 5.11 Renaming Selected Variables |

**Read**

Refer to the link below for more details and examples on the .rename() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌rename.html>

#### 1.2.5 Creating Dummy Variables

As mentioned, categorical variables must be converted to dummy variables before they can be evaluated and included in the computation of the scikit-learn algorithms. Dummy variables are binary variables that only have two values: 0 and 1. If an observation belongs to a certain category, the corresponding dummy variable will be 1, otherwise 0. Since each category of a categorical variable will be transformed to a dummy variable, the number of categories has indeed a direct impact on the number of dummy variables in the final DataFrame used in the algorithm. As a result, it is important to keep the number of categories at a rather low level. The syntaxes and procedures for category reduction have been discussed in Chapter 1.2.2 and 1.2.3 of this study unit.

In Python, we can convert categorical variables to dummy variables using the get\_dummies() function of the pandas package.

|  |
| --- |
| DataFrame\_name["column"] = pd.get\_dummies(data, drop\_first) |

The parameter drop\_first is used to instruct Python to take the first category as the reference level and remove it from the resulting DataFrame. The reason to define a reference level for categorical variables and to remove it from the DataFrame is to avoid linear dependence in the data matrix, which causes error in the calculation. The default setting here is drop\_first = False. In this case, all dummy variables will remain in the resulting DataFrame. Since most of the modules in scikit-learn have their own algorithms to deal with this issue, we can keep this setting without causing error in the estimation process.

The meaning of the parameter data is obvious. Note that pandas will create dummy variables for each uniquely existing string in all non-numeric variables. If we have a numeric categorical variable which we would like to convert to dummy variables as well, we will need to change its data type using the .astype() method.

|  |
| --- |
| DataFrame\_name.astype({"var\_name": "type\_str", …}) |

A dictionary should be assigned to the .astype() method with the variable names as the keys and the data types as the values, which can be "int", "float", "category", "str", "bool", etc. To create dummy variables for a numeric categorical variable, we can either change the type of the orginal variable to "category" or "str".

|  |
| --- |
| **Example (Cont’d):** For the categorical variables workclass, marital-status, education, race and gender, dummy variables must be created before they can be included in the construction of the analytics models.    Figure 5.12 Creating Dummy Variables from Categorical Variables  In Figure 5.12, the numeric variables remain unchanged as they do not have the appropriate data type to be converted. The categorical variables, on the other hand, have now been replaced by the dummy variables entirely. Therefore, we recommend carrying out such transformation in a new DataFrame so that we do not lose the original data in case anything goes wrong. |

**Read**

Refer to the link below for more details and examples on the get\_dummies() function of the pandas package:

[https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get\_dummies‌.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html)

**Read**

Refer to the link below for more details and examples on the .astype() method of the pandas package:

[https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame. ‌astype.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.%20‌astype.html)

#### 1.2.6 Data Transformation

In the previous sections, we have discussed how to prepare categorical variables for scikit-learn analytics algorithms. Though numeric variables are generally easier to deal with, they may also cause trouble in the model estimation. For instance, their range of values can be rather wide. As a result, variables with such characteristic tend to have higher impact in the model than those with smaller value ranges. Hence, they need to be scaled down to match the value range of the other numeric variables. The most common methods for this purpose are normalisation and standardisation.

In Chapter 5.4 of Study Unit 4, we have discussed how to execute log-transformation, standardisation, and normalisation on pandas DataFrames. However, we have had to construct our own programs for standardisation and normalisation since they are not included in pandas. In scikit-learn, on the other hand, these functions can be found in the preprocessing module, where the normalize() function is actually straightforward and easy to handle.

|  |
| --- |
| Object\_name = sklearn.preprocessing.normalize(X) |

The object assigned to parameter X can be a DataFrame, a NumPy array, etc.

Conversely, the syntax for standardisation requires the initialisation of the estimator first, the data can then be transformed in the subsequent step.

|  |
| --- |
| scaler = preprocessing.StandardScaler()  scaler.fit(X)  Object\_name = scaler.transform(X) |

We first initiate scaler as the estimator object for StandardScaler from the preprocessing module. Then, the mean and standard deviation of the object X, which is usually an Array or a DataFrame, will be computed by the fit() function. In the final step, the object X will be standardised by the transform() function.

|  |
| --- |
| **Example (Cont’d):** In view of K-Means clustering that we will be carrying out in the next Chapter, we need to normalise the numeric variables. In particular, the values of the variables capital-gain and capital-loss have very wide range (0 - 99,999 and 0 - 4,356, respectively) and may affect the fit or the explanatory power of the model.  In the first step, we select those columns that should be normalised and save it in a new DataFrame. The reason of generating a subset DataFrame here is that normalize() will generate a function which transforms all numeric variables including the dummy variables created from the categorical data. Though their values will remain 0 and 1 after normalisation, we can shorten the processing time by not letting irrelevant variables involved in the process.    Figure 5.13 Selecting Numeric Variables for Normalisation  In the first line, we create a list of variables to be normalised and subset the original DataFrame using this list. Since the resulting object of the normalize() function is a NumPy array, we will have to convert it back to a pandas DataFrame with column and row labels (Note: NumPy arrays have no labels on both axes). Therefore, we need to save the labels as two Python lists named DF\_model\_toNorm\_colnam and DF\_model\_toNorm\_rownam for later use.    Figure 5.14 Normalising Numeric Variables  After the normalisation process, we can convert the array DF\_model\_NormArray resulting from the preprocessing.normalize() function to a pandas DataFrame by the pd.DataFrame() function. We can also set our own column and row labels by assigning the lists DF\_model\_toNorm\_colnam and DF\_model\_toNorm\_rownam to the columns and index parameters, respectively.  In the final step, we rename the normalised variables and append them to the original DataFrame.    Figure 5.15 Concatenating Normalised Variables with the Original DataFrame  Note that we have used the original variable names as the column labels of the normalised variables. If we simply concatenate these DataFrames, these labels would be duplicate. Hence, we need to append “\_norm” as suffix to all labels of the normalised variables by the .add\_suffix() method. Only after ensuring that all labels in the concatenated DataFrames are unique, we can merge the DataFrames. Furthermore, we have saved the names of the normalised variables with the suffix “\_norm” in a new list named numnormvar\_list just in case we will need to extract them again from the DataFrame in the future. |

**Read**

Refer to the link below for more details and examples on the normalize() function of the preprocessing module in the scikit-learn package:

<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.‌html>

**Read**

Refer to the link below for more details and examples on the sklearn.‌preprocessing.StandardScaler algorithm of the scikit-learn package:

<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.‌StandardScaler.html>

**Read**

Refer to the link below for more details and examples on the .add\_suffix() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌add_suffix.html>

#### 1.2.7 Splitting DataFrames for Training and Testing

In data analytics, the performance of a predictive model can be measured by its accuracy of predicting unseen data. However, such data are usually unavailable. On the other hand, testing the prediction performance of a model by applying it on the original data based on which the model was constructed in the first place is not sensible at all. Therefore, analysts usually “hold back” a subset of data, namely the testing dataset, for model evaluation purpose. The remaining data form the training dataset for the construction of the model.

In Python, the module model\_selection provides the train\_test\_split() function to draw observations randomly from the original DataFrame into the training and the testing datasets.

|  |
| --- |
| Object\_name = sklearn.model\_selection.train\_test\_split( arrays, test\_size, random\_state) |

The objects assigned to the parameter arrays can be NumPy arrays, pandas DataFrames, etc. A value between 0 and 1 should be given to the parameter test\_size, which determines the proportion of observations in the original array that should be distributed to the testing dataset. The default value here is 0.25. The parameter random\_state controls the shuffling of the data before applying the split. The default value here is "None", which means that different random seeds will be selected every time the function is being executed. That is, different observations will be chosen for the testing dataset in every run. Consequently, the testing result of the model will be different as well.

|  |
| --- |
| **Example (Cont’d):** After all the data preparation steps have been carried out, we can now split the DataFrame into a training and a testing dataset. The testing data proportion we require here is 30%, and we will set random\_state to “None”.    Figure 5.16 Training Dataset after Split from the Original DataFrame  We can see from Figure 5.16 that the training dataset contains 31,655 rows which are exactly 70% of the original 45,222 observations. Furthermore, the rows have been shuffled completely as indicated by the chaotic order of the row indices.  The remaining 13,567 rows are now assigned to the testing dataset as Figure 5.17 illustrates.    Figure 5.17 Testing Dataset after Split from the Original DataFrame |

**Read**

Refer to the link below for more details and examples on the train\_test\_split() function of the model\_selection module in the scikit-learn package:

[https://scikit-learn.org/stable/modules/generated/sklearn.model\_selection.train\_test\_‌split.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)

#### 1.2.8 Extracting Dependent and Independent Variables

In scikit-learn, the ultimate command of many algorithms to fit a model on a DataFrame is the .fit() function. The .fit() function has usually two parameters: X and Y, where Y could be optional for some algorithms. The parameter X is the design matrix that contains all independent variables, and Y is the vector of the target variable. Both X and Y can be NumPy arrays or pandas DataFrames. As a result, we need to extract the independent variables as a matrix and the dependent variable as a vector from the original DataFrame.

The procedure here is rather straightforward. We simply select the column that represents our dependent variable and save it as a new object.

|  |
| --- |
| y = DataFrame\_name["target\_var"] |

Similarly, the matrix of the independent variables can be selected in the same manner.

|  |
| --- |
| X = DataFrame\_name[["X1", "X2", …]] |

Note that it is required to wrap the names of the independent variables in a list (square brackets) first before putting them in the index operator [].

If the DataFrame only contains the independent variables and the target variable, we can also simply drop the target variable from the original DataFrame to obtain X.

|  |
| --- |
| X = DataFrame\_name.drop("target\_var") |

If the target variable is categorical and has been transformed to various dummy variables, the names of the dummy variables must be put in a list when passing them to the .drop() method.

|  |
| --- |
| **Example (Cont’d):** Before separating our training and testing datasets for the dependent and independent variables, we shall save the row indices of both the datasets in two new lists for later identification of the corresponding observations.    Figure 5.18 Saving the Row Indices in Python Lists  Since our DataFrames DF\_model\_train and DF\_model\_test only contain the independent and dependent variables, we can simply extract the dependent variables for y\_train and y\_test and drop them from the DataFrames subsequently to generate X\_all\_train and X\_all\_test.  Nevertheless, the target variable income has been a binary categorical variable with the levels “<=50K” and “>50K”. Hence, there must be two dummy variables in the DataFrame, one for each of the categories. For model construction, however, we only need one of them, and the decision here favours the dummy variable “income\_>50K” where 1 represents observation with income more than 50,000 USD and 0 the opposite. Through this value assignment, the natural order of the income can be reflected by the order of values in this dummy variable, and we can avoid confusion when interpreting the modelling results.    Figure 5.19 Slicing Training and Testing Datasets for the Independent and the Target Variables  In the first line, we create a list called y\_dummyvar with the names of the two dummy variables in it. We can use this list to remove the corresponding columns in X\_all\_train and X\_all\_test. For the construction of y\_train and y\_test, the column with the label that matches the second item in y\_dummyvar will be selected from the training and testing DataFrames.  Note that both X\_all\_train and X\_all\_test still contain the original numeric variables as well as their normalised counterparts. We keep both sets purposely so that we can respond to the requirements of the different algorithms flexibly by choosing either set of them. This is also the reason why we have created two lists named numvar\_list and numnormvar\_list in the previous sections so that we can select the columns directly from the DataFrames in the future.  From Figure 5.20 and Figure 5.21, we can see that y\_train and y\_test have been converted to two pandas Series with 31,655 and 13,567 elements, respectively. Figure 5.22 and Figure 5.23 show the training and testing DataFrames of the independent variables X\_all\_train and X\_all\_test.    Figure 5.20 Training Dataset for the Target Variable    Figure 5.21 Testing Dataset for the Target Variable    Figure 5.22 Training Dataset for the Independent Variables    Figure 5.23 Testing Dataset for the Independent Variables  When applying unsupervised machine learning algorithms where we do not need to partition our DataFrame into training and testing datasets, we can split the dependent variables and the input variables directly from the original DataFrame DF\_model\_final. The mechanism is just the same as Figure 5.19. The corresponding syntaxes and outputs are given in Figure 5.24 and Figure 5.25.    Figure 5.24 Dataset for the Dependent Variable    Figure 5.25 Dataset for the Independent Variables |

## Chapter 2 Clustering

### 2.1 Introduction of K-Means Clustering

Lesson Recording - Introduction and Fitting of K-Means Clustering by scikit-learn

Clustering is a multivariate analytics technique to group “similar” observations into finite number of disjoint clusters. One of the most popular clustering algorithms is the K-Means method. This technique is very efficient in clustering large data sets. The algorithm here is to split the data into K groups with equal variance by minimising the variation within the cluster. This variation is called the inertia or within-cluster sum-of-squares. In other words, “members” in the same cluster should be as “similar” as possible, while observations from different clusters should be most distinguishable.

Different from some other clustering algorithms in which the number of clusters will only emerge during the grouping process, the K-Means method requires the number of clusters to be specified before the algorithm starts. The clusters are characterised by their centroids, which can be interpreted as the centre of an area in a two-dimensional space, and are hence the average of all the observations within a cluster. As the name of the algorithm suggests, there should be K different means (centroids) and they should be explored during the clustering process.

The process of K-Means clustering can be described in five main steps:

1. K observations are randomly selected as initial cluster centroids where K is a pre-defined positive integer.
2. Compute the distance of each object to the centroids.
3. Based on the distance computed, each object is assigned to the nearest centroid. Objects assigned to the same centroid form a cluster. There will be K different clusters.
4. For each cluster, recompute the centroid using the objects assigned to the cluster. The iteration starts again from Step 2.
5. The iteration stops when the centroids remain unchanged or a specified number of iterations has been performed.

Note that the distance mentioned in 2) refers to the Euclidean distance in general. The Euclidean distance between an object and a cluster centroid is measured by the sum of the squared differences between the values of some selected clustering criteria, which are usually some input variables of the object, and the values of the same clustering criteria of the centroid.

Subsequent to the clustering process, it is important to make sure that the resulting clusters really create some insights. To interpret the clusters, the characteristics of each cluster should be explored by looking at the summary statistics (e.g. mean, min, max) of the clustering criteria. A good clustering solution should allow us to describe the profile of each cluster clearly.

In addition, there are objective measures for evaluating the quality of clustering solutions: cohesion, separation and parsimony. The cohesion measures the similarity of the objects in a cluster. This value should be small because these objects should be similar. The separation, on the other hand, measures how dissimilar the clusters are, and this value should be high. Here, we can apply the Silhouette coefficient since it combines both the cohesion and the separation. Briefly speaking, the Silhouette coefficient is a value between -1 and 1 that measures the relationship between the intra-cluster distances and nearest cluster distances. The mean of the individual Silhouette coefficients will be computed for every clustering solution for evaluation. A high and positive average Silhouette coefficient suggests appropriate and useful clustering solution. On the contrary, negative Silhouette coefficient indicates a rather undesirable clustering result.

Furthermore, parsimony is another important criterion in clustering. As a result, we prefer smaller number of clusters if the quality of the corresponding clustering solution is satisfactory. Nevertheless, the number of clustering criteria should also be parsimonious, so that the clustering solution can be interpreted conveniently.

### 2.2 Fitting K-Means Clustering by Scikit-Learn

In scikit-learn, all algorithms are controlled and executed by the so-called estimator. We can adjust our parameters for the modelling process in the syntax of the estimator declaration. In K-Means Clustering, the estimator is called KMeans. And it can be imported from the cluster module of the sklearn package.

|  |
| --- |
| from sklearn.cluster import KMeans |

First, we need to initiate the KMeans estimator and adjust the estimation parameters according to our needs.

|  |
| --- |
| km\_Object = sklearn.cluster.KMeans(n\_clusters = 8, init = "k-means++", n\_init = 10, max\_iter = 300, tol = 0.0001, precompute\_distances = "auto", random\_state = None) |

The following table provides description and explanation of the parameters.

Table 5.2 Parameters of the KMeans Estimator

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Value Type** | **Description** |
| n\_clusters  *(Default: 8)* | integer | The number of clusters to form as well as the number of centroids to generate. |
| init (*Default:  "k-means++")* | "k-means++", "random", callable, array-like of shape | Method for initialisation:  "k-means++": selects initial cluster centres for k-mean clustering in a smart way to speed up convergence.  "random": choose n\_clusters observations at random from the data as initial centroids.  Array: An array with number of rows equal to the n\_clusters and number of columns equal to the number of variables that give the initial centroids.  Callable: It should take arguments X, n\_clusters and a random state and return an initialisation. |
| n\_init *(Default: 10)* | integer | Number of times the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n\_init consecutive runs in terms of inertia. |
| max\_iter *(Default: 300)* | integer | Maximum number of iterations of the k-means algorithm for a single run. |
| tol  *(Default: 1e-4)* | float | Relative tolerance of the difference in the cluster centres of two consecutive iterations to declare convergence. |
| precompute\_ distances *(Default: "auto")* | "auto", True, False | Precompute distances (faster but takes more memory).  "auto": do not precompute distances if n\_samples \* n\_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.  True: always precompute distances.  False: never precompute distances. |
| random\_state *(Default: None)* | integer,  RandomState instance, None | Determines random number generation for centroid initialisation. Use an integer to make the randomness deterministic. |

(Source: [https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans. ‌html](https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.%20‌html))

Here are some explanations regarding some items mentioned in Table 5.2:

* A callable is a program part that can be called such as a user-defined function or a built-in method.
* Centroid seeds refer to the initial cluster centres.
* Inertia measures the within-cluster sum-of-squares and should be minimised in the clustering process.

Next, we can apply the KMeans estimator on a prepared DataFrame.

|  |
| --- |
| km\_fit\_Object = km\_Object.fit(X, sample\_weight = None) |

The parameter X is a prepared DataFrame based on which the clusters are constructed. With sample\_weight we can pre-specify the weights for each observation in X. If it is set to None, which is also the default here, all observations will be assigned equal weight.

The fitted estimator of the K-Means algorithm is saved in km\_fit\_Object. To obtain and view the results of the estimation, we still need to request scikit-learn to predict the cluster classification for each observation in the DataFrame X.

|  |
| --- |
| km\_pred\_Object = km\_Object.fit\_predict(X, sample\_weight = None) |

The parameters here are the same as the .fit() function. The parameter X contains the data for which the cluster prediction will be calculated. The pre-specified individual weights in sample\_weight will be assigned to all observations in the dataset. The output object here is an n-dimensional array of length n\_samples, i.e., the number of observations in the dataset. The items in the array are indices of the cluster that each sample belongs to.

As mentioned in Chapter 2.1, the number of clusters, K, must be specified before the clustering algorithm starts. One way to determine the optimal value of K is the elbow method. Elbow method is a popular technique that uses the inertia as the measurement to compare the distortions in some clustering solutions with different K. The distortion is the sum of squared distances of each data point to the centroids. The plot of distortions which looks like an arm will then be generated. The best value of K can be found at the “elbow”, the inflection point on the curve. To determine the inertia of a clustering solution, we can apply the .inertia\_ method on a KMeans estimator.

|  |
| --- |
| **Example (Cont’d):** Since K-Means clustering is an unsupervised machine learning algorithm, we do not need to split our data into a training and a testing dataset. We can use the entire available DataFrame for clustering purpose.    Figure 5.26 Create a DataFrame with Only Normalised Input Variables  Nevertheless, as mentioned in Chapter 1.2.6., it is more sensible to include normalised data to build the clusters since numeric variables with extreme value ranges such as capital-gain or capital-loss have to be scaled. In the first line, X\_km is the DataFrame with no normalised variables and is used for the elbow-test to determine the optimal number of clusters. The name of all the normalised variables are stored in the list numnormvar\_list (see Figure 5.15). In the second line, X\_km\_norm is the DataFrame with no non-normalised variables and will be used for the clustering process. The name of all the numeric variables before normalisation is stored in numvar\_list (see Figure 5.13). In X\_km\_norm, we rename the normalised variables back to their original variable names by removing their suffix “\_norm” in order to simplify their labels for later output.  To find out the best K, the number of clusters, we can conduct an elbow test. Here, we will compute the inertia of the K-Means clustering solutions that contains 1 to 7 clusters and compare their distortions.    Figure 5.27 Calculating Inertia for the Elbow Method  We use a for-loop here to run through all clustering solutions with 1 to 7 clusters. The parameter in the KMeans estimator is kept as simple as possible. We only set the number of clusters and instruct Python to select initial cluster centres for k-mean clustering in a smart way to speed up convergence by placing init = "k-means++". After fitting the K-Means clusters on our data, we store the corresponding inertia in the list named distortions for later use.  After the inertia are calculated, we can plot them for the Elbow method.    Figure 5.28 Elbow Method to Determine the Optimal Number of Clusters  In the first line of Figure 5.28, we use the matplotlib options to set the size and resolution of the chart. In the second line, we put the number of clusters in the clustering solutions on the x-axis and the inertia on the y-axis. From the shape of the graph, the elbow can be found at K = 2. As a result, we will use a 2-cluster solution in the following.    Figure 5.29 Fitting K-Means Clustering  The code in Figure 5.29 is basically identical to those in Figure 5.27 but without the loop. Here, we fix our number of clusters to 2 and the random\_state to 0, which enables us to reproduce the same clustering results in the future.    Figure 5.30 Predicting Classification of the Data  After creating the K-Means clusters based on our normalised numeric variables and the dummy variables of our categorical variables, we save the predicted cluster index of each observation in the array object named y\_pred. From the output, we can see that the cluster indices are 0 or 1. |

**Read**

Refer to the link below for more details and examples on the KMeans estimator, fit(), and fit\_predict() functions of the cluster module in the scikit-learn package:

<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>

### 2.3 Model Exploration and Evaluation

Lesson Recording - Explore & Evaluate K-Means Clustering Models by scikit-learn

As mentioned in the previous section, the characteristics of the clusters should be explored and interpreted. We can examine this by looking at some statistics such as mean, min, max, etc. of the clustering criteria based on which the clusters have been constructed. For this purpose, we can cross-tabulate the clustering criteria and the cluster index to understand the features of the clusters.

If a clustering criteria variable is categorical, our focus of interpretation will be on the proportional distribution of the clusters in each category. With the crosstab() function, which we have briefly introduced in Chapter 1.2.3, we can easily create a cross-table to fulfil our purpose.

|  |
| --- |
| pd.crosstab(index = criteria\_var, columns = cluster\_index, normalize = "index", margin = True) |

The object assigned to the first parameter is the row variable of the cross-tabulation. In most of the cases, one of the clustering criteria is placed here since the number of categories could be rather large, and it is more convenient to have them listed in rows rather than in columns. If multiple names are passed to this parameter, these variables will be tabulated in hierarchy. The second parameter controls the column variable. Our suggestion is to place the cluster index here since the number of clusters is usually limited. The parameter normalize can take the values "all", "index", "columns", {0, 1}, or {True, False} where the default value is False, or 0 equivalently. If normalize = True or 1, Python will return the table percentage of each cell, while the counts of each cell will be shown if normalize is False or 0. If normalize = "index", the row proportion of the cell will be calculated, while the column percentage will be provided if normalize = "column". If margin = True, the marginal frequency of the axis specified in the normalize will be listed out as well.

If a clustering criterion were numeric, the crosstab() function would not be a good choice since it would take every unique value in it as a separate category. In this case, we would rather let Python calculate some statistics of the clustering criteria for each cluster. And the .groupby() method of the pandas package, which has already been introduced in Chapter 5.3 of Study Unit 4, would become applicable in our Python program once again.

|  |
| --- |
| DF[[criteria\_var, cluster\_index]]  .groupby(by = [cluster\_index]).anymethod().transpose() |

The above syntax is specifically adjusted for clustering interpretation. First, we merge the selected clustering criteria variables from the original DataFrame with the cluster classification variable. The resulting DataFrame will be grouped by the cluster indices, and the method to compute the statistics of interest is appended to it. Note that the variable used in the .groupby() method, which is the cluster index in this case, will be displayed as rows. But we can transpose the result of the .groupby() method to swap the rows and columns. As a result, we put the cluster indices as columns in the same way crosstab() does. But it is more a step to standardise the presentation rather than an analytics requirement and hence optional for us to integrate it in our syntax or not.

|  |
| --- |
| **Example (Cont’d):** First, we convert the cluster index array to become a pandas DataFrame and name the variable “cluster”.    Figure 5.31 Converting the Cluster Index Array to pandas DataFrame  Subsequently, we concatenate the original DataFrame (before the dummy and normalised variables are created) and the cluster index variable.    Figure 5.32 Merging the Original DataFrame with the Cluster Index  With this DataFrame, we can create our cross tables with the appropriate labels.    Figure 5.33 Cross-Tabulation of income and the Predicated Classification  From the above table, we can see that the majority of the observations with income more than 50,000 USD are in cluster 0. Though this cluster also contains 62% of the low-income group, the proportion of the same income group is rather high in cluster 1, namely 38%. The first impression is that the observations in cluster 0 are financially more well-off.    Figure 5.34 Cross-Tabulation of workclass and the Predicated Classification  Figure 5.34 shows the cross-tabulation of workclass and the cluster index. Here, we can see that while cluster 0 contains most of the self-employed individuals (85% vs. 15%), the distributions of the other work classes correspond roughly to the distribution of the clusters in the entire sample (68% vs. 32%).    Figure 5.35 Cross-Tabulation of education and the Predicated Classification  The cross-tabulation of education and the cluster index does not provide a conclusive relationship between them. While individuals with primary education level are over proportionally represented in cluster 0, postgraduates share a similar proportion in this cluster. As a result, the clusters do not differentiate the individual education level at all.    Figure 5.36 Cross-Tabulation of race and the Predicated Classification  From this cross-tabulation, Afro-Americans are over proportionally represented in cluster 1 than other ethnic groups.    Figure 5.37 Cross-Tabulation of Numeric Variables and the Predicated Classification  From Figure 5.37, we can see that individuals in cluster 0 work averagely 6 hours more in a week and their ratio of capital-gain to capital-loss is much higher than those in cluster 1. Their average age, however, does not differ significantly in both clusters. |

In Chapter 2.1, we introduced the Silhouette coefficient as a measure to evaluate the cohesion and separation of a clustering solution. In scikit-learn, we can apply the following syntax to calculate it.

|  |
| --- |
| metrics.silhouette\_score(criteria\_var, cluster\_index) |

Note that the silhouette\_score() function is from metrics and not the KMeans module.

|  |
| --- |
| **Example (Cont’d):** To calculate the Silhouette coefficient, we need the clustering criteria array that is stored in the DataFrame X\_km\_norm and the cluster index array named y\_pred.    Figure 5.38 Calculating the Silhouette Coefficient  The closer the Silhouette coefficient is to 1, the better is the cohesion and separation of the cluster. Here, a coefficient of 0.24 is indeed not very promising. In other words, the similarity within the cluster and the dissimilarity among the clusters are not very clear in this clustering solution. This confirms the findings resulting from the previous cross-tabulations in which the clusters do not provide a clear differentiation of some of the clustering criteria as well. |

Another possibility to understand a clustering solution is the graphical approach. The idea of this approach is to plot all the data points with their cluster classification in a two-dimensional scatter plot. This approach would be rather straightforward if only one or two input variables have been used as clustering criteria. In the multivariate case, we need to reduce the dimensionality of all the input variables down to two before plotting.

One of the most common methods to reduce the dimensions of a high-dimensional array (or matrix) is the Principal Component Analysis (PCA). The idea of the PCA is to project each data point onto the first few principal components which contain the majority of the variation in the data. The loss of information caused by the omission of the remaining components is then insignificant and the data dimension is reduced to the number of principal components. In our case, we may only be interested in the first two components to span the space for our scatter plot.

|  |
| --- |
| pca\_Object = sklearn.decomposition.PCA(n\_components) |

Note that PCA() is an estimator from the decomposition module. With the above syntax, we can specify the number of components we would like the result to contain. To reduce the dimension an array by PCA(), we need to execute the following syntax.

|  |
| --- |
| pca\_fit\_Object = sklearn.decomposition.fit\_transform(criteria\_var) |

Here, we need to use the .fit\_transformation() instead of the .fit() function since we are interested in the transformed values of all the criteria variables. The returned object is an n-dimensional NumPy array with the transformed data. Hence, the array has 2 columns, and its number of rows corresponds to the number of observations in the DataFrame that has been passed to the .fit\_transformation() function in the first place.

|  |
| --- |
| **Example (Cont’d):** In the first step, we reduce the dimensionality of the clustering criteria array X\_km\_norm to 2 components.    Figure 5.39 Reducing the Dimensionality of the Input Variable DataFrame  The resulting values in the array X\_pca are the coordinates of each data point in the scatter plot.  In the next step, we separate the coordinates of observations of cluster 0 from those of cluster 1 since we would like to plot them in different colours in the final chart.    Figure 5.40 Selecting PCA Data Based on the Predicted Classification  To plot the data points of different clusters in different colours, we need two plt.scatter command since we have to fix the facecolour parameter in every line. The values on the x-axis are the values stored in the first column of the subset arrays of X\_pca called filtered\_label0 and filtered\_label1. The values in their second columns are the values on the y-axis.    Cluster 1  Cluster 0  Figure 5.41 Plotting of the K-Means Clustering Result  From Figure 5.41, we can see the clusters are formed based on the distance of their data points to the centroids. While the data points in the bottom left corner of the chart are grouped to cluster 0 (grey data points), those in the upper right corner belong to cluster 1 (cyan data points). The differentiation of the two clusters are indeed quite clear based on their “locations” in this two-dimensional scatter plot. However, both the cross-tabulations and the Silhouette coefficients indicate that the characterisation of the clusters by the clustering criteria is not as straightforward as this plot suggests. We can therefore suspect that information of some input variables has gone lost during the process of dimension reduction. |

**Read**

Refer to the link below for more details and examples on the .transpose() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌transpose.html>

Refer to the link below for more details and examples on the silhouette\_score function of the metrics module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette\_score. ‌html

Refer to the link below for more details and examples on the PCA estimator of the decomposition module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

## Chapter 3 Decision Trees

### 3.1 Introduction to Decision Trees

Lesson Recording - Introduction and Fitting of Decision Trees by scikit-learn

Decision trees are among the most common data mining methods which split a set (or subset) of observations to reach certain decision points based on some criteria eventually. Each decision point in the tree is also called a node and represents a subset of the sample based on which the decision tree is created. Nodes that are split from a superordinate node are called the child node while the origin node is called the parent node. A child node with no further subdivisions or splitting is called a leaf node.

Since each observation in the sample will be assigned to one of the nodes eventually, decision trees is a classification technique to separate a sample into multiple classes. The decision tree algorithm predicts the individual classification based on the values of some input variables and calculates the predicted value of the target variable at the same time. These rules of decision form the resulting model which can then be illustrated by a tree-like structure graphically. This structure convenes the interpretation of the modelling result.

We can also use the decision tree model to understand the relationship between the target variable and the input variable. In fact, decision tree handles complex relationship such as non-linearity and interaction rather well. Note that not all input variables are of the same importance in the classification process. Their hierarchy in the decision rules reflects in the decision tree in which input variables appear higher up are more important.

As mentioned, nodes without further splitting are called the leaf nodes. Their value is the prediction of the target variable for those observations classified in the corresponding nodes. If the target variable is categorical, the value of the leaf node will be the mode, the most frequent class. And it will be the mean of the data in that node if the decision tree is predicting a numeric variable.

Over the years, there have been many algorithms for splitting the nodes and hence the construction of the tree developed, proposed, and implemented in software packages. The most common ones are CHAID (chi-square automatic interaction detection), C5.0 (a proprietary algorithm) and CART (classification and regression tree). In Python, the estimator DecisionTreeClassifier of the scikit-learn package uses an optimised version of the CART algorithm.

As the name suggests, the CART algorithm is capable to create both classification trees and regression trees. While regression trees estimate the values of a continuous target variable, classification trees predict the outcome of a categorical target variable. In other words, CART is applicable on almost every type of output variables.

In CART, every parent node only has two child nodes. That is, CART will only split the tree into two sub-samples at every decision point. And the calculation of the split is based on the input variables that are also used to predict the target output. As the split process advances, the sample will be divided into more and more, smaller and smaller subsets. These subsets will also become more and more homogeneous. The whole splitting process will be terminated once certain stopping criteria are fulfilled.

The homogeneity of each subset reflects the split quality from a parent node to its child nodes. In classification tree, the homogeneity is measured by Gini and Entropy. Roughly speaking, both Gini and entropy measure the impurity of a node, but with different theoretical background. By comparing the impurity decrease across all possible splits in all input variables, the split with the highest reduction of impurity will be chosen. In scikit-learn, both Gini and entropy are options of the criterion parameter in the DecisionTreeClassifier.

In regression tree, the impurity is measured by the sum of squared error (SSE). The SSE is the total deviance of each observation from the sample mean. For each potential split, CART computes the SSE for each child node and the split with the lowest sum of SSE across all child nodes will be chosen. Since the mean of the target variable of the leaf node is equal to the predicted value of the target variable, splits with low SSE have child nodes that contain data whose target values are close to the mean value.

One possibility to stop the CART algorithm is when the impurity improvement of a new split drops below a certain pre-defined threshold. For instance, if a node has reached a rather low Gini or SSE respectively, meaning that the parent node itself is already rather homogeneous, another split from the node would only create homogeneous child nodes. In this case, this split is not really necessary since it does not decrease the impurity significantly.

Nevertheless, choosing the right thresholds to stop the split algorithm is not as straightforward as it seems. If the thresholds are high, the resulting tree could be oversimplified as splits become more difficult. Low thresholds, on the other hand, could lead to overcomplicated trees that are difficult to interpret and deploy.

Another possibility to stop the algorithm is when the tree has attained a pre-specified depth. The depth of the tree refers to the number of splits in it. This method can simply control the size of the tree without oversimplifying or overcomplicating it. One last stop criterion is to set a lower bound of observations in the nodes. Once the number of observations in all nodes has reached the bound, a new split from any node would only create child nodes that contain less observations than the lower bound allows. As a result, the lower bound blocks the algorithm from carrying out another split and the entire process ends.

One way to evaluate the performance of a decision tree is to examine the precision of its prediction. In other word, the predicted values of the target variable will be compared with the observed data. For classification trees, we can use the confusion matrix in which the correct and incorrect classifications are summarised. The larger the proportion of observations for which the predicted and observed classifications are identical, the more accurate is the decision tree model. For regression trees, the Root-Mean-Square-Error (RMSE) is usually used to measure the prediction accuracy of the model. Basically, the RMSE is kind of an average deviance of all the predicted values from their observed counterparts. The lower such deviance is, the closer are the predictions to the actual values, and the better is the model.

Furthermore, the performance of a decision tree to predict unseen data must be evaluated as well. For this purpose, we partition the original dataset randomly into a training and a testing dataset. As described in Chapter 1, the training dataset is used to construct the model while the predictive performance of the model will be evaluated based on the testing dataset. In other words, the decision tree as a predictive model is evaluated by its ability to apply what it has “learned” from the training data on the testing data. If the prediction accuracy of the model on the training data is much higher than the testing data, the model tends to be overfitted. It is too specialised to the structure of the training dataset and not generalised enough for other data that do not contain certain unique characteristics of the training data.

### 3.2 Fitting Decision Trees

Same as K-Means clustering, scikit-learn also has an estimator for decision trees model which is integrated in the tree module. In order to be able to use all possible functions in the module, the sklearn.tree module shall be imported in the first place.

|  |
| --- |
| from sklearn import tree |

If we want to apply a classification tree in Python, we have to initiate a DecisionTreeClassifier estimator object whereas if the data should be fitted by a regression tree, a DecisionTreeRegressor estimator object should be declared.

|  |
| --- |
| tree\_Object = sklearn.tree.DecisionTreeClassifier(criterion = "gini", splitter = "best", max\_depth = None, min\_samples\_split = 2, min\_samples\_leaf = 1, min\_weight\_fraction\_leaf = 0.0, max\_features = None, random\_state = None, max\_leaf\_nodes = None, min\_impurity\_decrease = 0.0, min\_impurity\_split = None, class\_weight = None) |
| tree\_Object = sklearn.tree.DecisionTreeRegressor(criterion = "mse", splitter = "best", max\_depth = None, min\_samples\_split = 2, min\_samples\_leaf = 1, min\_weight\_fraction\_leaf = 0.0, max\_features = None, random\_state = None, max\_leaf\_nodes = None, min\_impurity\_decrease = 0.0, min\_impurity\_split = None) |

From the above syntaxes, we can see the main differences between the estimators DecisionTreeClassifier and DecisionTreeRegressor are the values of the parameter criterion and the availability of the parameter class\_weight. The following table provides description and explanation of the parameters.

Table 5.3 Parameters of the DecisionTreeClassifier and DecisionTreeRegressor Estimators

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Value Type** | **Description** |
| criterion  *(Default: "gini" for classification, "mse" for regression)* | Classification: "gini", "entropy"  Regression: "mse", "friedman\_mse", "mae", "poisson" | The function to measure the quality of a split. For classification trees, the supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.  For regression trees, the supported criteria are "mse" for the mean squared error which is equal to variance reduction as feature selection criterion, "friedman\_mse", which uses mean squared error with Friedman’s improvement score for potential splits, "mae" for the mean absolute error, and "poisson" which uses reduction in Poisson deviance to find splits. |
| splitter (*Default: "best")* | "best", "random" | The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split. |
| max\_depth *(Default: None)* | integer | The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min\_samples\_split samples. |
| min\_samples\_ split  *(Default: 2)* | Integer or float | The minimum number of samples required to split an internal node:  If integer, consider min\_samples\_split as the minimum number.  If float, min\_samples\_split is a fraction and ceil(min\_samples\_split \* n\_samples) are the minimum number of samples for each split. |
| min\_samples\_ leaf  *(Default: 1)* | Integer or float | The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min\_samples\_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.  If integer, consider min\_samples\_leaf as the minimum number.  If float, min\_samples\_leaf is a fraction and ceil(min\_samples\_leaf \* n\_samples) are the minimum number of samples for each node. |
| min\_weight\_ fraction\_leaf *(Default: 0.0)* | float | The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample\_weight is not provided. |
| max\_features *(Default: None)* | Integer, float, "auto", "sqrt", "log2" | The number of features to consider when looking for the best split:  If integer, consider max\_features features at each split.  If float, max\_features is a fraction and int(max\_features \* n\_features) features are considered at each split.  If "auto",  max\_features = sqrt(n\_features)  If "sqrt",  max\_features = sqrt(n\_features)  If "log2",  max\_features = log2(n\_features).  If None,  max\_features = n\_features.  Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max\_features features. |
| random\_state *(Default: None)* | integer,  RandomState instance, None | Controls the randomness of the estimator. The features are always randomly permuted at each split, even if splitter is set to "best". When max\_features < n\_features, the algorithm will select max\_features at random at each split before finding the best split among them. But the best found split may vary across different runs, even if max\_features = n\_features. That is the case, if the improvement of the criterion is identical for several splits and one split has to be selected at random. To obtain a deterministic behaviour during fitting, random\_state has to be fixed to an integer. |
| max\_leaf\_nodes  *(Default: None)* | integer | Grow a tree with max\_leaf\_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. |
| min\_impurity\_ decrease  *(Default: 0.0)* | float | A node will be split if this split induces a decrease of the impurity greater than or equal to this value. |
| min\_impurity\_ split  *(Default: 0)* | float | Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf. |
| class\_weight *(Default: None)*  *Note: Only available for classification trees.* | dictionary,  list of dictionaries,  "balanced" | Weights associated with classes in the form {class\_label: weight}. If None, all classes are supposed to have weight one. For multi-output, a list of dictionaries can be provided in the same order as the columns of y.  Note that for multioutput, weights should be defined for each class of every column in its own dictionary.  The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data.  For multi-output, the weights of each column of y will be multiplied. |

(Source: [https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTree‌Classifier.html#sklearn.tree.DecisionTreeClassifier), <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html>)

Next, we can apply the DecisionTreeClassifier and DecisionTreeRegressor estimators on the prepared training DataFrames with the input variables X and the target variable y.

|  |
| --- |
| tree\_fit\_Object = tree\_Object.fit( X\_train, y\_train,  sample\_weight = None) |

For both the DecisionTreeClassifier and DecisionTreeRegressor estimators, scikit-learn also facilitates the parameter sample\_weight to specify individual weights in X\_train. If it is set to None, the default value, all observations will be assigned equal weight. In addition, sample\_weight will be multiplied with class\_weight if it is specified in the DecisionTreeClassifier estimator for classification trees.

The fitted estimator of the decision trees algorithm is saved in tree\_fit\_Object. We can now predict the classification of the data stored in the testing DataFrame saved in X\_test.

|  |
| --- |
| tree\_pred\_Object = tree\_Object.predict(X\_test) |

The returned object tree\_pred\_Object is a NumPy array which contains the predicted target values of every observation in the testing dataset. The number of rows here is therefore equivalent to the number of rows in X\_test.

|  |
| --- |
| **Example (Cont’d):** Before constructing the decision tree, we have to remove the normalised numeric variables first since their original variables are preferred.    Figure 5.42 Create a DataFrame without the Normalised Input Variables  In the next step, we initiate an object named clf for the classification tree estimator DecisionTreeClassifier since our target variable is binary. The decision tree model will then be fitted by the .fit() function using the training dataset.    Figure 5.43 Fitting Decision Trees  After the model has been created, the classification of the target variable will be predicted for the observations of both the training and testing datasets. The prediction accuracy on the training dataset is an indicator of the goodness of fit of the model while its predictive power can be assessed by its prediction accuracy on the testing dataset.    Figure 5.44 Predicting Classification of Training and Testing Datasets  The predicted classification of the testing dataset is stored in the NumPy array y\_pred while y\_class contains the predicted classification of the training data. |

**Read**

Refer to the link below for more details and examples on the

DecisionTreeClassifier estimator, the fit() and the predict() functions of the tree module in the scikit-learn package:

<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.‌html>

Refer to the link below for more details and examples on the

DecisionTreeRegressor estimator, the fit() and the predict() functions of the tree module in the scikit-learn package:

<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.‌html>

### 3.3 Model Evaluation

Lesson Recording - Evaluate Decision Trees by scikit-learn

As described in Chapter 3.1, there are various possibilities to evaluate the performance of a decision tree. One of them is the confusion matrix which compares the actual with the predicted classification of the sample. In Python, the confusion matrix can be computed by the function confusion\_matrix() from the metrics module.

|  |
| --- |
| metrics.confusion\_matrix(target\_var, tree\_pred\_Object) |

The object target\_var is the column of the target variable in the original DataFrame and tree\_pred\_Object is the resulting NumPy array from the predict() function of the DecisionTreeClassifier or DecisionTreeRegressor estimator.

Other indicators such as accuracy, precision, and recall scores can also be considered for assessing the predictive performance of a decision tree.

|  |
| --- |
| metrics.accuracy\_score(target\_var, tree\_pred\_Object) |
| metrics.precision\_score(target\_var, tree\_pred\_Object) |
| metrics.recall\_score(target\_var, tree\_pred\_Object) |

The above three measures are particularly useful in examining binary classification target variables. A binary target variable has two classes: 0 = “negative” and 1 = “positive”. Accuracy score measures the sample proportion that has been classified as positive and negative correctly. Precision score, on the other hand, also called the positive predicted value (PPV) represents the sample proportion that has been predicted as positive correctly (true positive) in relation to all the cases that are predicted as positive, regardless of their actual status. Recall score, or sensitivity, is the proportion of the true positive cases in relation to the actually positive sample.

|  |
| --- |
| **Example (Cont’d):** We will first calculate the accuracy, precision and recall scores for the predicted classification of the training dataset.    Figure 5.45 Prediction Performance on Training Data  The result seems very promising since the overall accuracy score is 0.92. Furthermore, over 92% of the observations classified in the income group with more than 50,000 USD p.a. belong really to that income class, and 75% of the individuals in the income class “>50K” are really predicted as such. As a result, the goodness of fit of the model is rather high.  Subsequently, we evaluate the predictive power of our decision tree model on the unseen data.    Figure 5.46 Prediction Performance on Testing Data  The overall accuracy here is worse than the results of the training data prediction but remain rather high (0.82). The precision score of the prediction on the testing data is, however, on a slightly lower level (66%), while the recall score is significantly worse than the classification prediction for the training data (54%).  Another method to evaluate the predictive performance of a decision tree model is the confusion matrix.    Figure 5.47 Confusion Matrix  Note the accuracy, precision and recall scores can all be found in the confusion matrix. The diagonal elements are the number of observations that have been classified to their own income categories accurately. This proportion is equivalent to the accuracy score calculated above: 82%. From the individuals in the “>50K” income category (3,329), only 1,814 (54%) have been classified correctly. This proportion is equal to the recall score. There are altogether 2,737 out of 13,567 observations classified in the “>50K” income category, and for 1,814 of those is the prediction accurate. The proportion of 66% is exactly the precision score. |

The most important tool to understand and to evaluate a decision tree is the plot of the tree itself. The tree module of the scikit-learn package provides the plot\_tree() function to generate such a graph conveniently.

|  |
| --- |
| tree.plot\_tree(decision\_tree, max\_depth = None, feature\_names = None, class\_names = None, label = "all", filled = False, impurity = True, node\_ids = False, proportion = False, rounded = False, precision = 3, ax = None, fontsize = None) |

The following table provides description and explanation of the parameters.

Table 5.4 Parameters of the plot\_tree Function

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Value Type** | **Description** |
| decision\_tree  *(No default value)* | An decision tree regressor or classifier object | The decision tree to be plotted. |
| max\_depth *(Default: None)* | integer | The maximum depth of the representation. If None, the tree is fully generated. |
| feature\_names  *(Default: None)* | List of strings | Names of each of the features (variables). If None, generic names will be used ("X[0]", "X[1]", …). |
| class\_names  *(Default: None)* | List of strings or Boolean | Names of each of the target classes in ascending numerical order. Only relevant for classification and not supported for multi-output. If True, shows a symbolic representation of the class name. |
| label  *(Default: "all")* | "all", "root", "none" | Whether to show informative labels for impurity, etc. Options include "all" to show at every node, "root" to show only at the top root node, or "none" to not show at any node. |
| filled *(Default: False)* | Boolean | When set to True, paint nodes to indicate majority class for classification, extremity of values for regression, or purity of node for multi-output. |
| impurity *(Default: True)* | Boolean | When set to True, show the impurity at each node. |
| node\_ids  *(Default: False)* | Boolean | When set to True, show the ID number on each node. |
| proportion  *(Default: False)* | Boolean | When set to True, change the display of ”values” and/or “samples” to be proportions and percentages respectively. |
| rounded *(Default: False)* | Boolean | When set to True, draw node boxes with rounded corners and use Helvetica fonts instead of Times-Roman. |
| precision *(Default: 3)* | integer | Number of digits of precision for floating point in the values of impurity, threshold and value attributes of each node. |
| ax *(Default: None)* | matplotlib axis | Axes to plot to. If None, use current axis. Any previous content is cleared. |
| fontsize *(Default: None)* | integer | Size of text font. If None, determined automatically to fit figure. |

(Source: <https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html>)

The plot\_tree() function can also be combined with the matplotlib options such as the plot size, the borderline settings, etc. to optimise the output of the tree.

|  |
| --- |
| **Example (Cont’d):** Before plotting the decision tree, we need to prepare the labels of the input variables and the categories of the target variables. While we can use the .columns.values method to extract the variables names in X\_train, we can only get the category names of the target variable by applying the .unique() method to extract all the unique values or strings in it.    Figure 5.48 Preparing Labels for Tree Plot  Finally, the decision tree can be plotted by the tree.plot\_tree() function.    Figure 5.49 Plotting Decision Tree with tree.plot\_tree  For the parameter feature\_names, we can assign the array X\_label prepared in the previous step to it since it requires the labels of all the input variables, including the dummy variables resulting from the categorical variables. The parameter class\_name requires the labels of the categories in the target variable, which are now stored in the array y\_label. We set the fontsize parameter to 3 and omit the impurity statistics (impurity = False) in each node so that the displayed information are still complete and readable. To visually distinguish the nodes, we instruct Python to fill the node boxes with colours (filled = True) while the corners do not need to be rounded (rounded = False). In order to keep the decision tree chart within a certain size, we limit the depth of the tree to a maximum of 4 levels (max\_depth = 4).    Figure 5.50 Decision Tree Plot  From Figure 5.50, we can see that the first split of the tree is generated by the marital status “Married”. The “non-married” sub-sample will be assigned to the child node on the left and those “married” observations to the child node on the right. In the next step, both nodes are split by the estimated threshold values of the variable capital-gain. These values are different in the two child nodes. The split of the tree will then continue until certain stop criteria are fulfilled. Since we limit the tree depth down to the fourth split level in our chart, nodes of the further split levels will only be displayed in grey boxes with no proper information in it. |

**Read**

Refer to the link below for more details and examples on the confusion\_matrix() functions of the metrics module in the scikit-learn package:

[https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion\_matrix‌.html](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html)

Refer to the link below for more details and examples on the accuracy\_score() functions of the metrics module in the scikit-learn package:

[https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy\_score‌.html](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html)

Refer to the link below for more details and examples on the precision\_score() functions of the metrics module in the scikit-learn package:

[https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision\_score‌.html](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html)

Refer to the link below for more details and examples on the recall\_score() functions of the metrics module in the scikit-learn package:

<https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html>

Refer to the link below for more details and examples on the plot\_tree() functions of the tree module in the scikit-learn package:

<https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html>

## Summary

In this unit, we have seen how Python can be used to carry out analytics tasks based on two techniques: k-means clustering and decision trees. One of the most common packages in Python for data analytics and machine learning algorithms is scikit-learn. In scikit-learn, the analytics algorithm is called an estimator, and its parameters need to be calibrated before the fitting process are carried out. Once the calibration step is completed, we can apply the model on prepared DataFrames.

However, the available DataFrames and their contents are usually not in the format and shape that the scikit-learn algorithms require in the first place. Therefore, we need to prepare the dataset accordingly. In this unit, we have learned how to remove missing data, reduce categories, discretise numeric variables, select and rename variables, transform data, partition data into training and testing datasets and extracting dependent and independent variables from the original DataFrame.

Subsequently, the application of K-Means clustering and decision trees have been demonstrated. To understand and evaluate the classification and prediction results of the K-Means clustering, we need to generate cross-tables of the clustering criteria and the cluster indices, calculate the Silhouette coefficient, and plot the data points with optical differentiation regarding their classification on a two-dimensional scatter plot. For decision tree modules, we can create tree plots to understand the classification process and compute the confusion matrix, the scores of accuracy, precision as well as recall to assess their predictive performance.

## References

pandas (n.d.). *pandas.crosstab*. The pandas development team. <https://pandas.pydata.org/docs/reference/api/pandas.crosstab.html>

pandas (n.d.). *pandas.DataFrame.add\_suffix*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.add_suffix.html>

pandas (n.d.). *pandas.DataFrame.astype*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.astype.html>

pandas (n.d.). *pandas.DataFrame.rename*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.rename.html>

pandas (n.d.). *pandas.DataFrame.replace*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame‌.replace.html>

pandas (n.d.). *pandas.DataFrame.transpose*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.transpose.html>

pandas (n.d.). *pandas.get\_dummies*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌get_dummies.html>

scikit learn (n.d.). *Clustering*. scikit-learn developers.  
<https://scikit-learn.org/stable/modules/clustering.html>

scikit learn (n.d.). *Decision Trees*. scikit-learn developers.  
<https://scikit-learn.org/stable/modules/tree.html>

scikit learn (n.d.). *Installing scikit-learn*. scikit-learn developers.  
<https://scikit-learn.org/stable/install.html>

scikit learn (n.d.). *sklearn.cluster.KMeans*. scikit-learn developers.   
<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>

scikit learn (n.d.). *sklearn.decomposition.PCA*. scikit-learn developers.  
<https://scikit->learn.org/stable/modules/generated/sklearn.decomposition.PCA.‌html

scikit learn (n.d.). *sklearn.metrics: Metrics*. scikit-learn developers.  
<https://scikit-learn.org/stable/modules/classes.html?highlight=metrics#module-sklearn.metrics>

scikit learn (n.d.). *sklearn.metrics.accuracy\_score*. scikit-learn developers.  
<https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_‌score.html>

scikit learn (n.d.). *sklearn.metrics.confusion\_matrix*. scikit-learn developers. <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_‌matrix.html>

scikit learn (n.d.). *sklearn.metrics.precision\_score*. scikit-learn developers.  
<https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_‌score.html>

scikit learn (n.d.). sklearn.metrics.recall\_score. scikit-learn developers.  
<https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_‌score.html>

scikit learn (n.d.). sklearn.metrics.silhouette\_score. scikit-learn developers. <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_‌score.html>

scikit learn (n.d.). sklearn.model\_selection.train\_test\_split. scikit-learn developers. <https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.‌train_test_split.html>

scikit learn (n.d.). sklearn.preprocessing.normalize. scikit-learn developers. <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.‌normalize.html>

scikit learn (n.d.). sklearn.preprocessing: Preprocessing and Normalization. scikit-learn developers.  
[https://scikit-learn.org/stable/modules/classes.html?highlight=preprocessing#‌module-sklearn.preprocessing](https://scikit-learn.org/stable/modules/classes.html?highlight=preprocessing#module-sklearn.preprocessing)

scikit learn (n.d.). sklearn.preprocessing.StandardScaler. scikit-learn developers. <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.‌StandardScaler.html>

scikit learn (n.d.). sklearn.tree.DecisionTreeClassifier. scikit-learn developers. <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTree‌Classifier.html>

scikit learn (n.d.). sklearn.tree.DecisionTreeRegressor. scikit-learn developers. <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTree‌Regressor.html>

scikit learn (n.d.). sklearn.tree.plot\_tree. scikit-learn developers.  
<https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html>

scikit learn (n.d.). User Guide. scikit-learn developers.  
<https://scikit-learn.org/stable/user_guide.html>

## Formative Assessment

1. What is an estimator in scikit-learn?
2. It is the estimation algorithm of a data analytics/machine learning module.
3. It is the function to fit a model.
4. It is a parameter to calibrate the model estimation.
5. It is a module of the scikit-learn package.
6. What type of tasks do most of the functions in the metrics module carry out?
   1. They convert imperial measurements (inch, foot, mile) to metric measurements (centimetre, metre, kilometre).
   2. They compute the mean distance of the data to the centroids in K-Means clustering.
   3. They compute metrics and distances for the evaluation of classification performance.
   4. They solve inequalities in geometry.
7. Why is partitioning data into a training and a testing dataset necessary for supervised machine learning?
   1. We can assess the predictive power of a model by applying it on unseen data.
   2. We can increase the goodness of fit of a model by creating a testing dataset.
   3. We can increase the learning ability of the algorithm.
   4. We have then two datasets to construct different models as alternatives.
8. Which function or method can be used to reduce the number of categories in a categorical variable?
   1. .get\_dummies()
   2. normalize()
   3. .deletecat()
   4. .replace()
9. In which of the following syntaxes will the mean and standard deviation of an object X be calculated during a standardisation process?
   1. processing.standardize()
   2. scaler = preprocessing.StandardScaler()
   3. scaler.fit(X)
   4. scaler.transform(X)
10. Whenever random numbers have to be drawn in scikit-learn, there is a parameter named random\_state included in the function. What does random\_state actually control?
    1. It draws a random number from an interval [-1, 1].
    2. It creates a variable in your DataFrame to store all random numbers that have been drawn since the first run of the program.
    3. It controls the probability distribution of the random numbers.
    4. It draws the same “random numbers” in every run to make the results reproducible.
11. Which function or method can be helpful to determine the optimal number of clusters?
    1. .transpose()
    2. .inertia\_
    3. .silhouette\_score()
    4. .PCA()
12. How does the Principal Component Analysis reduce the dimension of an array?
    1. It projects the data in the array onto a few principal components without losing too much variation in it.
    2. It deletes those variables with a pairwise correlation larger than 0.5.
    3. It removes the insignificant variables based on a regression model.
    4. It selects randomly two variables and calculate the Silhouette coefficient. The pair of variables with the highest coefficients form the principal components.
13. What is not a potential stop criterion in a decision tree construction process?
    1. The tree depth
    2. The number observations in the nodes
    3. The impurity improvement
    4. The accuracy score
14. Which of the following indicators can give us some information regarding the predictive performance of a decision tree?
    1. precision score
    2. Gini and entropy
    3. Silhouette coefficient
    4. SSE and RMSE

## Suggested Solutions

1. What is an estimator in scikit-learn?
2. **It is the estimation algorithm of a data analytics/machine learning module.**

**Correct. The estimator in scikit-learn contains the entire algorithm of an analytics module.**

1. It is the function to fit a model.

*Incorrect. We need to apply the .fit() function on the estimator object to fit a model.*

1. It is a parameter to calibrate the model estimation.

*Incorrect. The estimator is the entire algorithm and not just the parameters in it.*

1. It is a module of the scikit-learn package.

*Incorrect. An estimator is part of a module of the scikit-learn package.*

1. What type of tasks do most of the functions in the metrics module carry out?
   1. They convert imperial measurements (inch, foot, mile) to metric measurements (centimetre, metre, kilometre).

*Incorrect. There are no such functions in the metrics module.*

* 1. They compute the mean distance of the data to the centroids in K-Means clustering.

*Incorrect. The calculation of such (Euclidean) distances is integrated in the KMeans module.*

* 1. **They compute metrics and distances for the evaluation of classification performance.**

**Correct. The functions in metrics are particularly useful in evaluating classification models.**

* 1. They solve inequalities in geometry.

*Incorrect. There are no such functions in the metrics module.*

1. Why is partitioning data into a training and a testing dataset necessary for supervised machine learning?
   1. **We can assess the predictive power of a model by applying it on unseen data.**

**Correct. Testing the prediction performance of a model by applying it on the data based on which the model was constructed is not sensible at all. Hence, we need a new dataset for this purpose.**

* 1. We can increase the goodness of fit of a model by creating a testing dataset.

*Incorrect. The goodness of fit of a model cannot be increased by partitioning a dataset.*

* 1. We can increase the learning ability of the algorithm.

*Incorrect. The learning ability cannot be enhanced by partitioning a dataset.*

* 1. We have then two datasets to construct different models as alternatives.

*Incorrect. Different models can also be constructed by the same dataset.*

1. Which function or method can be used to reduce the number of categories in a categorical variable?
   1. .get\_dummies()

*Incorrect. The .get\_dummies() method creates dummy variables from a categorical variable.*

* 1. normalize()

*Incorrect. The .normalize() function normalises numeric variable.*

* 1. .deletecat()

*Incorrect. There is no method called .deletecat() in the scikit-learn package.*

* 1. **.replace()**

**Correct. With the .replace() method, we replace the labels of various categories by one label, and the number of categories is hence reduced.**

1. In which of the following syntaxes will the mean and standard deviation of an object X be calculated during a standardisation process?
   1. processing.standardize()

*Incorrect. There is no such syntax in the scikit-learn package at all.*

* 1. scaler = preprocessing.StandardScaler()

*Incorrect. This line initiates the StandardScaler() estimator.*

* 1. **scaler.fit(X)**

**Correct. This line computes the mean and standard deviation of the object X.**

* 1. scaler.transform(X)

*Incorrect. This line standardises the data in X.*

1. Whenever random numbers have to be drawn in scikit-learn, there is a parameter named random\_state included in the function. What does random\_state actually control?
   1. It draws a random number from an interval [-1, 1].

*Incorrect.* *The random\_state parameter does not control the range from which the random numbers are drawn.*

* 1. It creates a variable in your DataFrame to store all random numbers that have been drawn since the first run of the program.

*Incorrect. There is no storage or record of the random numbers drawn in Python.*

* 1. It controls the probability distribution of the random numbers.

*Incorrect. The random\_state parameter does not control the underlying distribution of the random numbers at all.*

* 1. **It draws the same “random numbers” in every run to make the results reproducible.**

**Correct. If we assign an integer to random\_state, Python will produce the same set of “random numbers” in every run. Different integers result in different sets of “random numbers”.**

1. Which function or method can be helpful to determine the optimal number of clusters?
   1. .transpose()

*Incorrect. The .transpose() method swaps rows and columns of an array.*

* 1. **.inertia\_**

**Correct. Inertia measures the distortions in some clustering solutions which are used by the Elbow method. The “elbow” of the distortion plot is the optimal number of clusters.**

* 1. .silhouette\_score()

*Incorrect. The Silhouette coefficient measures the cohesion and separation of clusters and is not used to determine the optimal number of clusters.*

* 1. .PCA()

*Incorrect. The PCA is a technique to reduce the dimensionality and not used to determine the optimal number of clusters.*

1. How does the Principal Component Analysis reduce the dimension of an array?
   1. **It projects the data in the array onto a few principal components without losing too much variation in it.**

**Correct. The omission of the remaining, insignificant components is the core idea of dimension reduction by the PCA.**

* 1. It deletes those variables with a pairwise correlation larger than 0.5.

*Incorrect. The PCA does not delete variables directly.*

* 1. It removes the insignificant variables based on a regression model.

*Incorrect. The PCA does not require regression models for dimension reduction.*

* 1. It selects randomly two variables and calculate the Silhouette coefficient. The pair of variables with the highest coefficients form the principal components.

*Incorrect. The Silhouette coefficient measures the cohesion and separation of clusters and is not used to reduce dimensionality.*

1. What is not a potential stop criterion in a decision tree construction process?
   1. The tree depth

*Incorrect. We can indeed stop the split of a decision tree if the tree has reached a certain depth.*

* 1. The number observations in the nodes

*Incorrect. If the sub-sample in any child node is smaller than the minimum required number of observations after a new split, the estimator will terminate the tree construction before the split.*

* 1. The impurity improvement

*Incorrect. The impurity improvement is actually an important stop criterion of a decision tree construction process. If new splits of a decision tree do not improve its impurity, the algorithm will stop further splitting of nodes.*

* 1. **The accuracy score**

**Correct. The accuracy score is an indicator for the predictive performance of a decision tree and not a stop criterion.**

1. Which of the following indicators can give us some information regarding the predictive performance of a decision tree?
   1. **precision score**

**Correct. The precision score represents the sample proportion that has been predicted as positive correctly (true positive) in relation to all the cases that are predicted as positive, regardless of their actual status.**

* 1. Gini and entropy

*Incorrect. Gini and entropy measure the homogeneity of the nodes in a decision tree.*

* 1. Silhouette coefficient

*Incorrect. The Silhouette coefficient measures the cohesion and separation of clusters.*

* 1. SSE and RMSE

*Incorrect. SSE and RMSE are measurements of impurity for regression trees.*