dataMining02-data_exploration-adults

February 12, 2020

1 From UCI Machine Learning Repository

1.1 Adult dataset

This data file does not have a header with column names. Look at the ".names" text file in the Data Folder and use the same procedure used for Iris

Print als the types of the columns using the `types` attribute

[2]:
```
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
plt.style.use('ggplot')
```

names = ['age','workclass','fnlwgt','education','education-num','marital-status','occupation' ,'relationship','race','sex','capital-gain','capital-loss','hours-per-week','native-country', 'high-income']

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data'

Load the data in the dataframe `df` and then show the column types with the `.dtypes` attribute of a Pandas DataFrame

[3]:
```
names =␣
 ↪['age','workclass','fnlwgt','education','education-num','marital-status','occupation'
        ␣
 ↪,'relationship','race','sex','capital-gain','capital-loss','hours-per-week','native-country
 ↪'high-income']
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.
 ↪data'
df = pd.read_csv(url, sep = ',', names = names
#                 , index_col = False # since the first column is integer,␣
 ↪without this
                                    # it would be interpreted as row label
                )
print(df.dtypes)
```

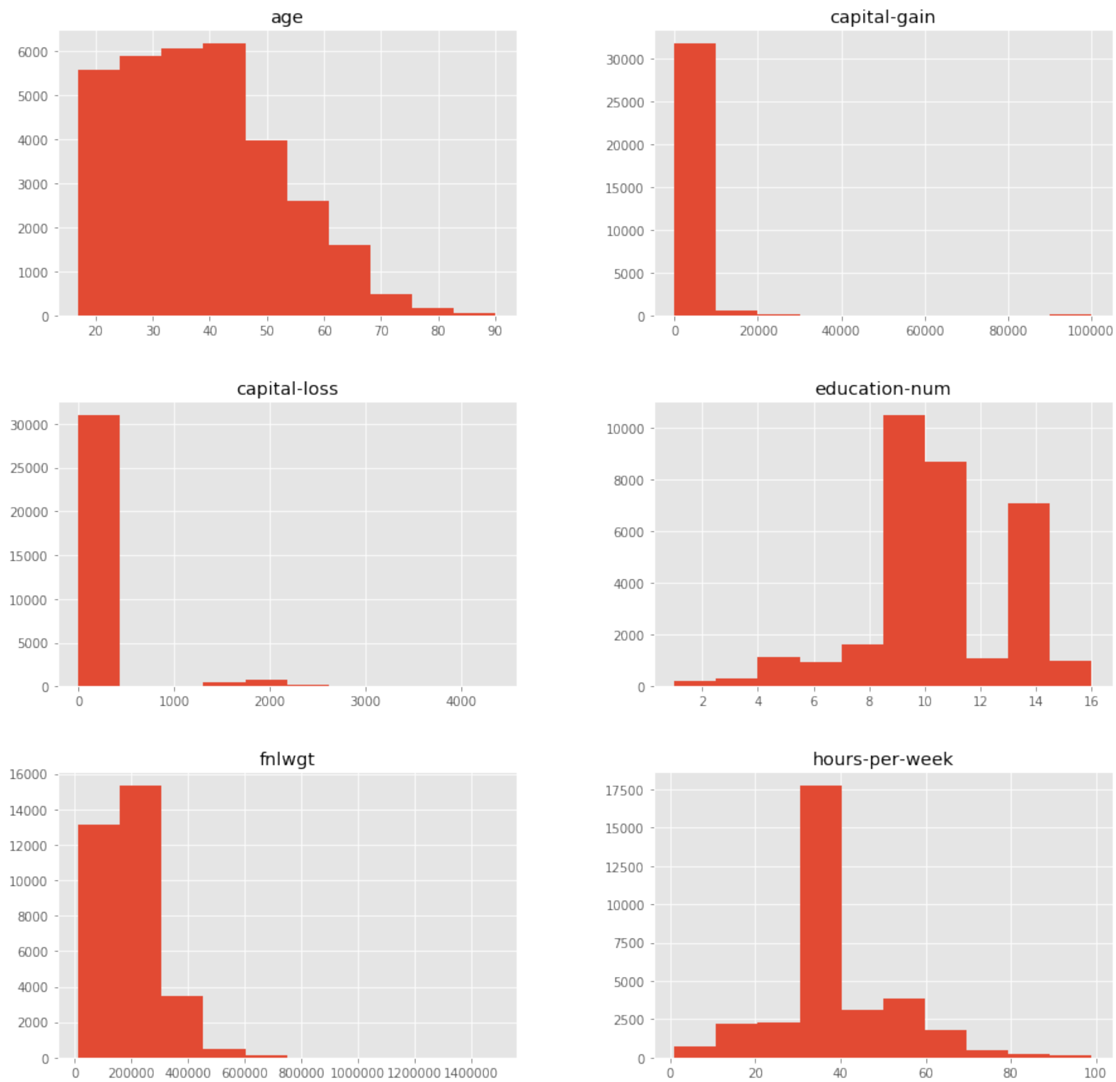
```
age                int64
workclass         object
fnlwgt             int64
education         object
education-num      int64
marital-status    object
occupation        object
relationship      object
race              object
sex               object
capital-gain       int64
capital-loss       int64
hours-per-week     int64
native-country    object
high-income       object
dtype: object
```

Show the head and then generate the histograms for all the columns

[4]:
```python
df.head()
```

[4]:
```
   age         workclass  fnlwgt  education  education-num  \
0   39         State-gov   77516  Bachelors             13
1   50  Self-emp-not-inc   83311  Bachelors             13
2   38           Private  215646    HS-grad              9
3   53           Private  234721       11th              7
4   28           Private  338409  Bachelors             13

       marital-status         occupation   relationship   race     sex  \
0       Never-married       Adm-clerical  Not-in-family  White    Male
1  Married-civ-spouse    Exec-managerial        Husband  White    Male
2            Divorced  Handlers-cleaners  Not-in-family  White    Male
3  Married-civ-spouse  Handlers-cleaners        Husband  Black    Male
4  Married-civ-spouse     Prof-specialty           Wife  Black  Female

   capital-gain  capital-loss  hours-per-week native-country high-income
0          2174             0              40  United-States       <=50K
1             0             0              13  United-States       <=50K
2             0             0              40  United-States       <=50K
3             0             0              40  United-States       <=50K
4             0             0              40           Cuba       <=50K
```

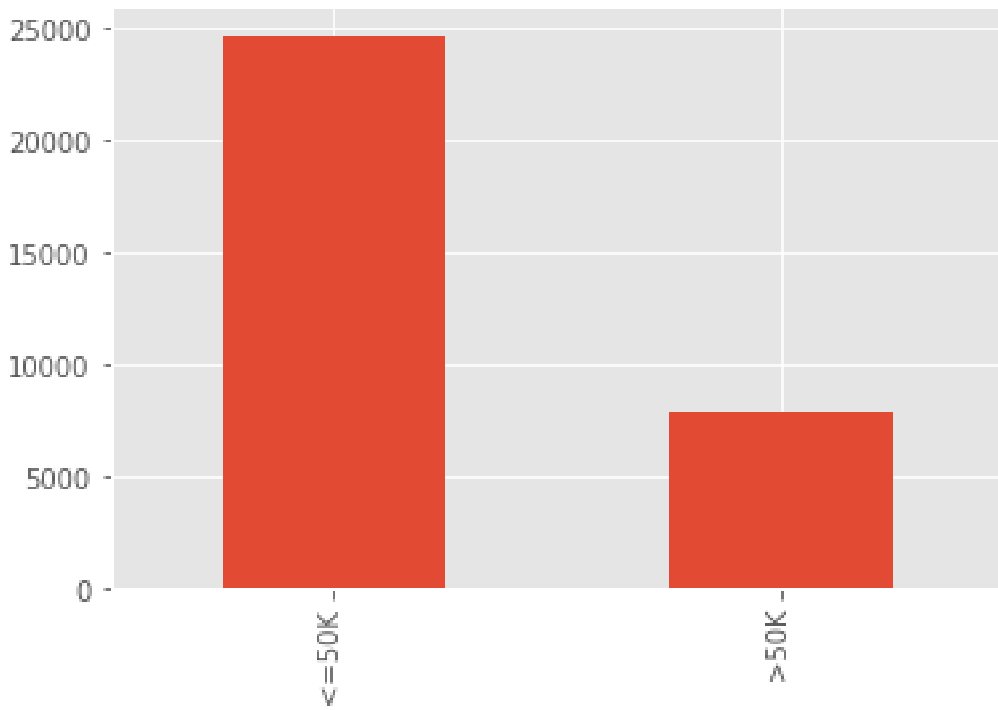
[13]:
```python
pd.DataFrame.hist(df, figsize = [15,15]);
```



Show a bar graph with the value counts of the attribute `high-income`. Use the method `value_counts` of Pandas, then `plot` with the option `kind = 'bar'`

[15]:
```
df['high-income'].value_counts().plot(kind = 'bar')
```

[15]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x228c9a0f5c0>
```



1.1.1 More examples of figures

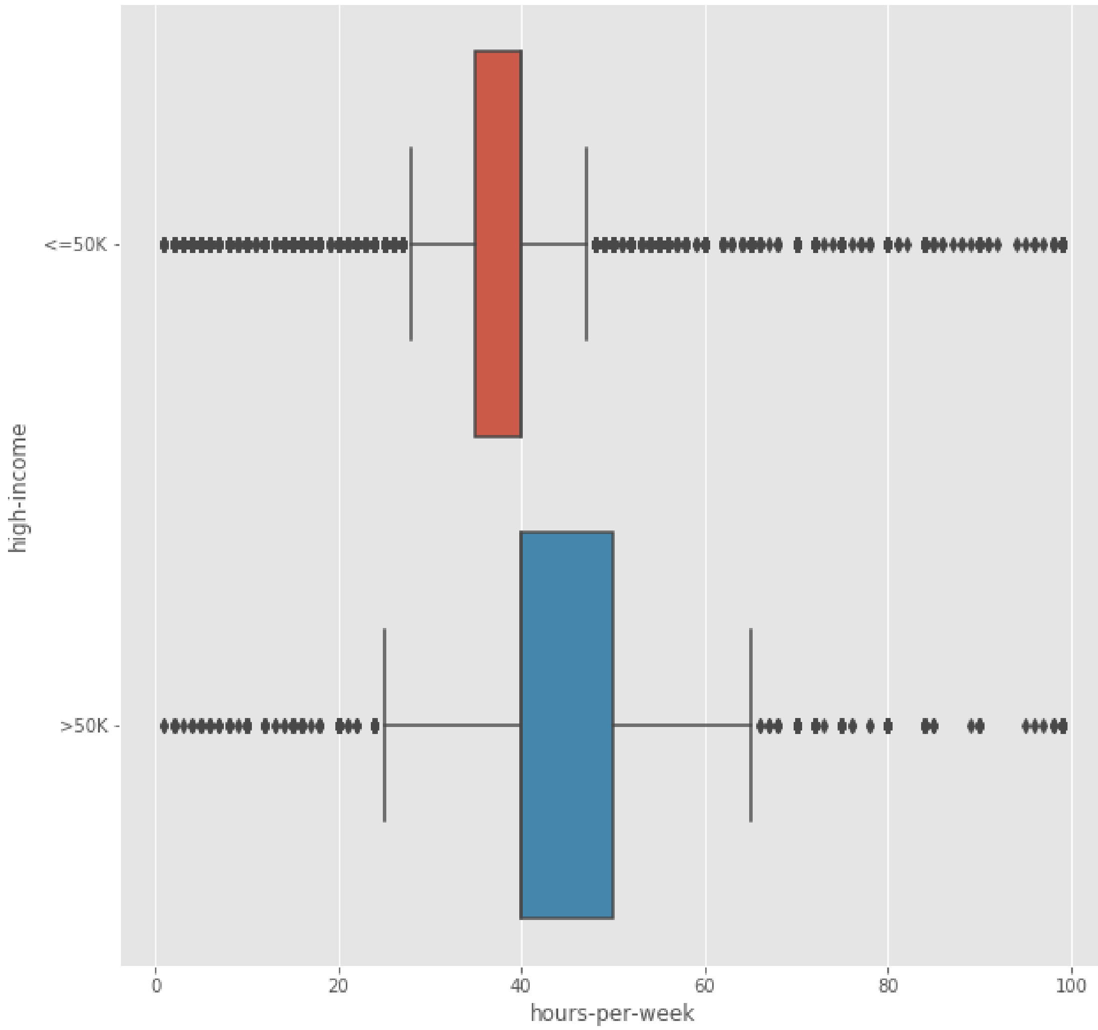
Boxplot

More on boxplots

Use the `boxplot` method of Seaborn with `hours-per-week` in the `x` axis and `high-income` in the `y` axis. The columns are extracted with the `loc` method of Pandas DataFrames, with index expression `[:,'attribute-name']` (means all the elements of column `attribute-name`)

[16]:
```
plt.figure(figsize = [10,10])
sns.boxplot(x=df.loc[:,'hours-per-week'], y=df.loc[:,'high-income'])
```

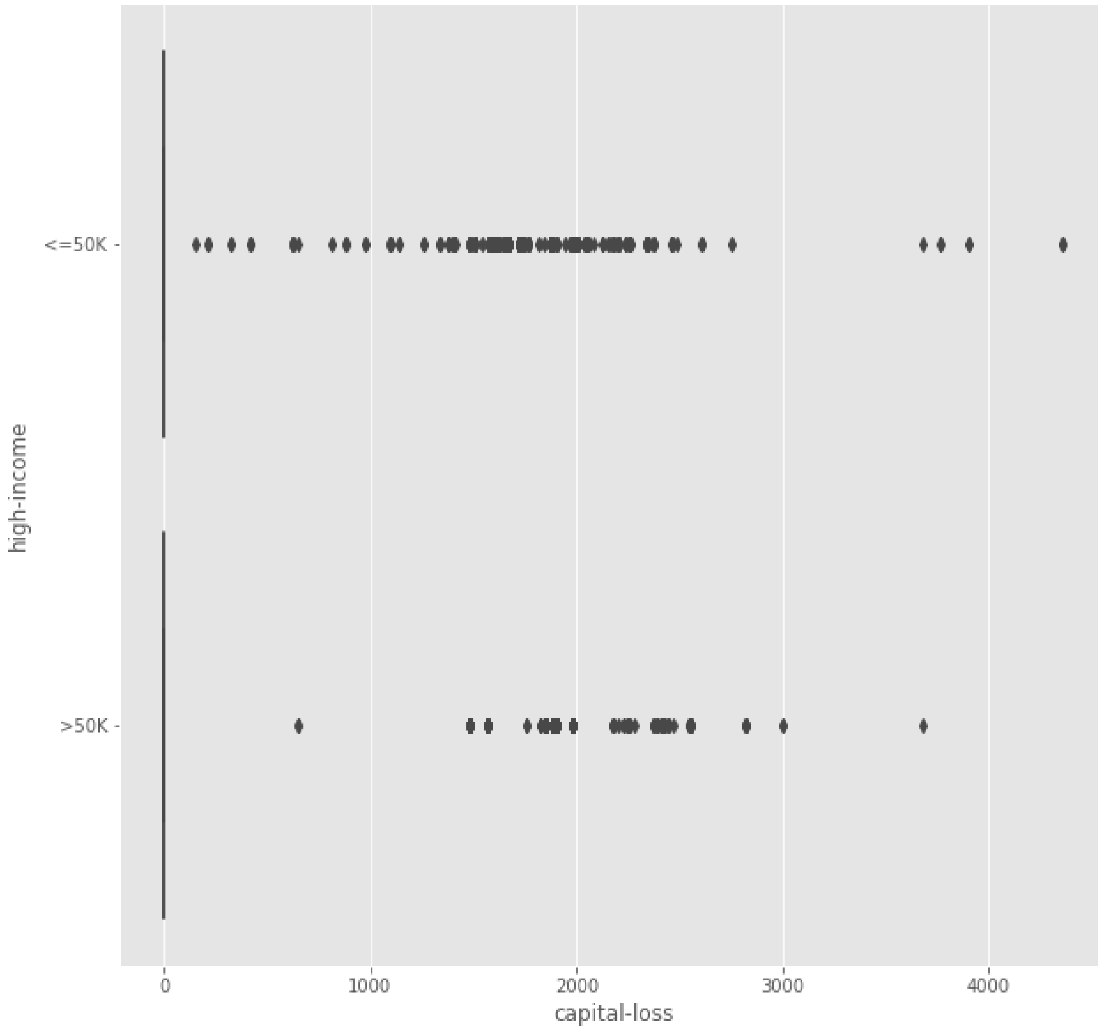
[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x228c99a1240>
```



Similar boxplot for 'capital-loss' and 'high-income']

[17]:
```
plt.figure(figsize = [10,10])
sns.boxplot(x=df.loc[:,'capital-loss'], y=df.loc[:,'high-income'])
```

[17]: <matplotlib.axes._subplots.AxesSubplot at 0x228c99576a0>



Something is wrong, the figure does not look like a proper boxplot.

Let's look at the **capital-loss** column with the `describe` method

[18]:
```
df['capital-loss'].describe()
```

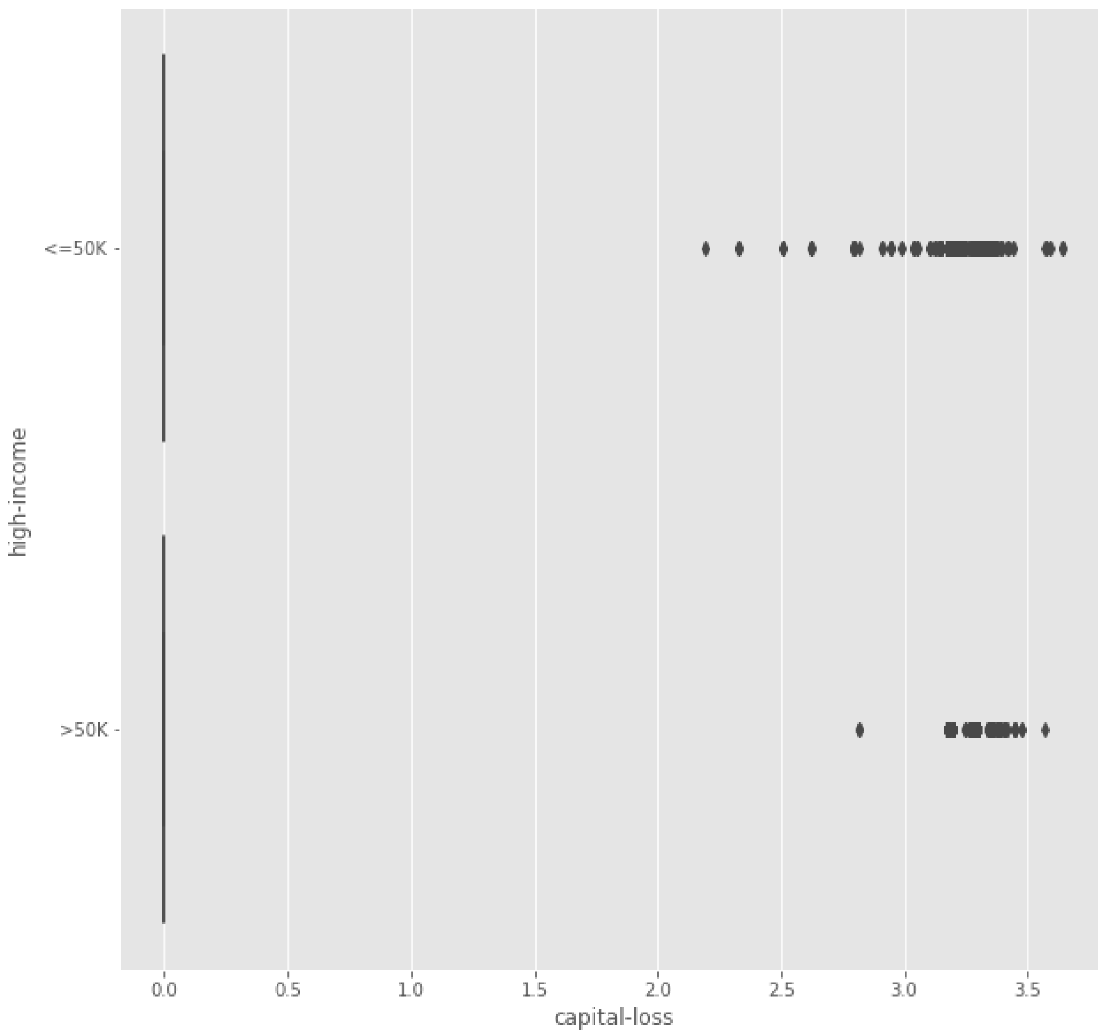
[18]:
```
count    32561.000000
mean        87.303830
std        402.960219
min          0.000000
25%          0.000000
50%          0.000000
75%          0.000000
max       4356.000000
Name: capital-loss, dtype: float64
```

The three quartiles are all zero, and there are no left outliers.

Let's try with a logarithmic transformation (add +1 to deal with the zero values) - use the `log10` function of `numpy` to transform the `capital-loss`+1 - prepare a plot figure of size [10,10] - boxplot with Seaborn

[19]:
```
from numpy import log10
plt.figure(figsize = [10,10])
sns.boxplot(x=log10(df.loc[:,'capital-loss']+1), y=df.loc[:,'high-income'])
```

[19]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x228c98b95c0>
```

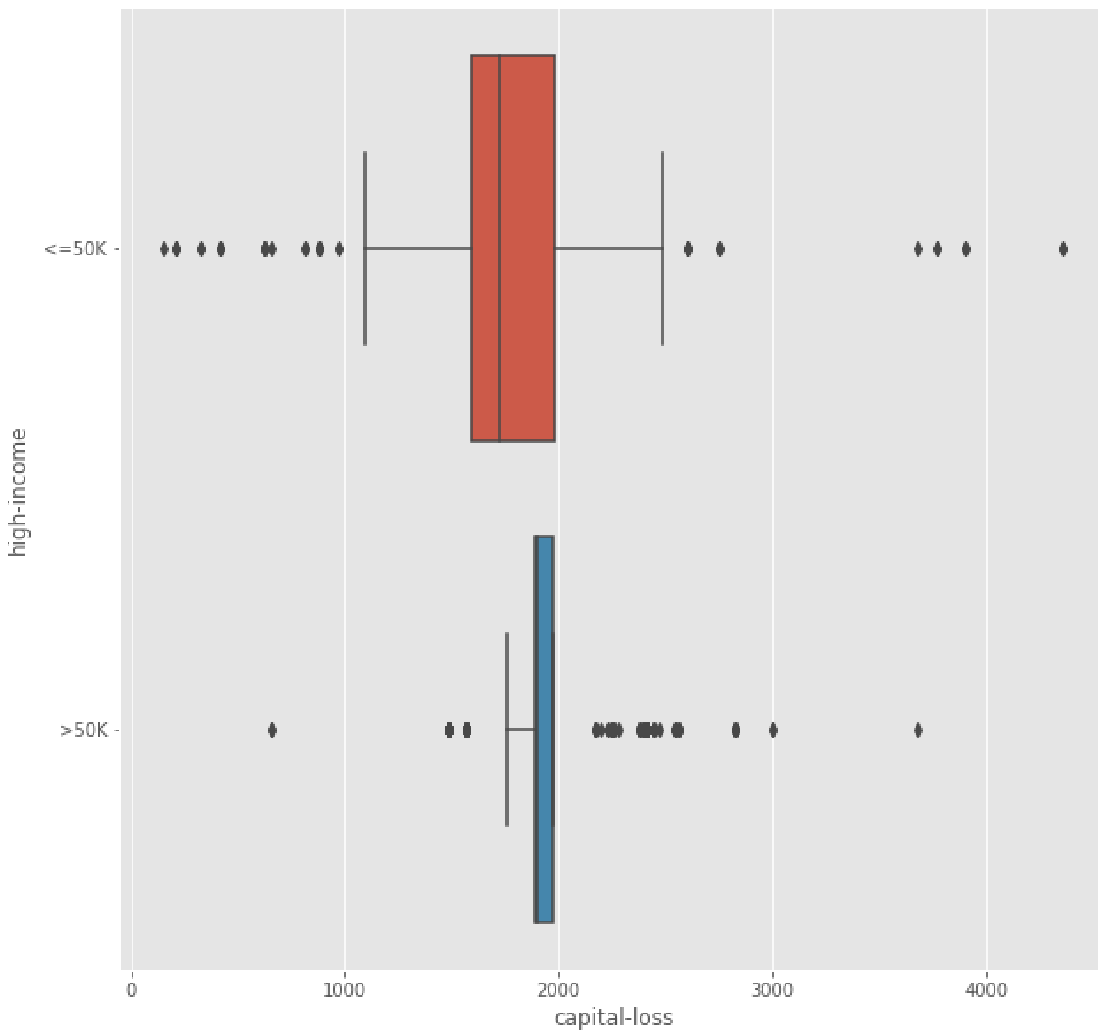


You can observe that a most of the data are 'compressed' at 0 - it is due to the zero values to which we added 1, whose log is 0 again

Look at the rows with non-zero values: in the x values, instead of the `:` indicating 'all the rows' we must use a 'selector expression', in this case `df['capital-loss']!=0`

[20]:
```
plt.figure(figsize = [10,10])
sns.boxplot(x=df.loc[df['capital-loss']!=0,'capital-loss'], y=df.loc[:
 ↪,'high-income'])
```

[20]: <matplotlib.axes._subplots.AxesSubplot at 0x228ca1afc50>



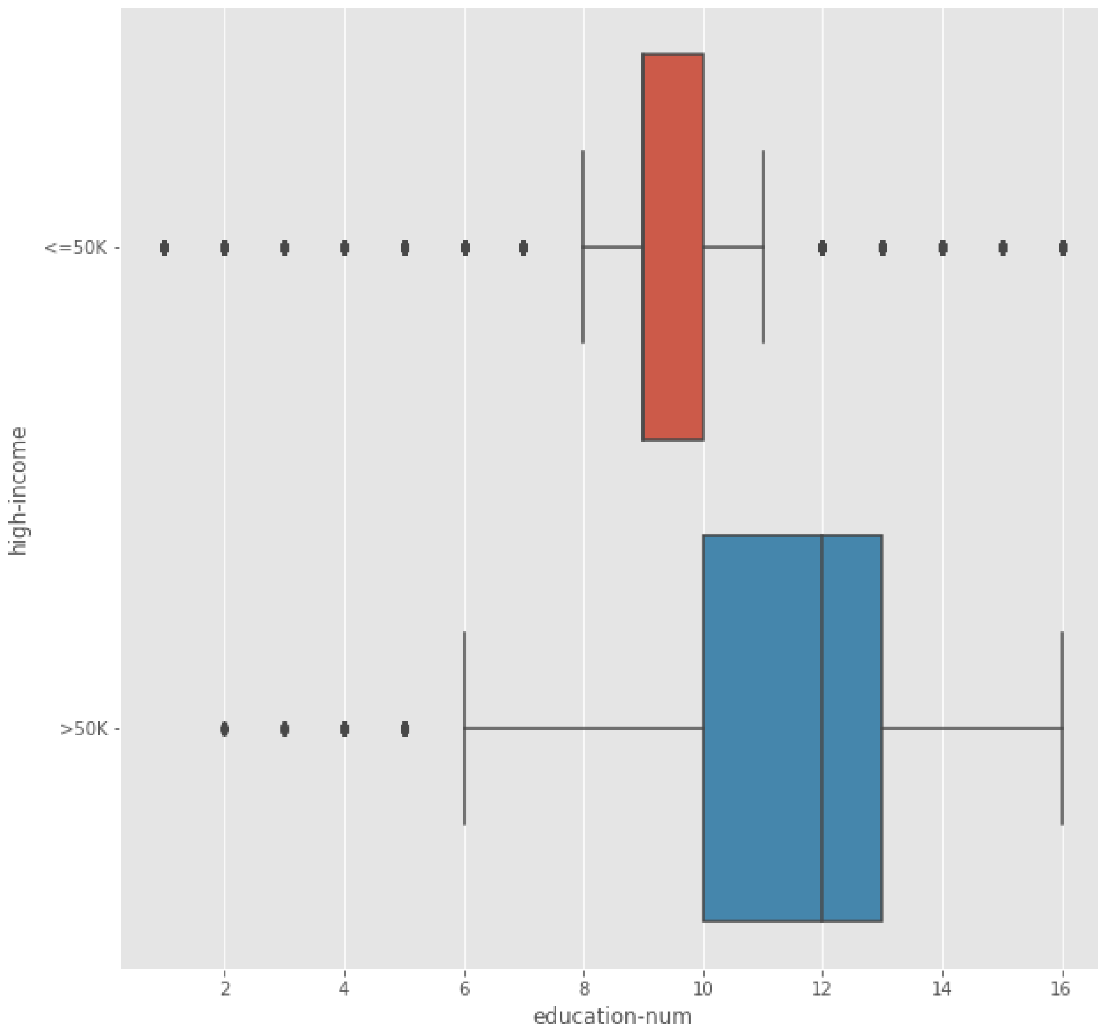
Now we see that the non-zero values have some structure

1.1.2 Plot another pair of columns

`education-num` and `high-income`

[21]:
```
plt.figure(figsize = [10,10])
sns.boxplot(x=df.loc[:,'education-num'], y=df.loc[:,'high-income'])
```

[21]: <matplotlib.axes._subplots.AxesSubplot at 0x228c9ac1240>



dataMining02-data_exploration-iris

February 12, 2020

1 From UCI Machine Learning Repository

[1]:
```
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
plt.style.use('ggplot')
```

1.1 Iris Dataset

This data file does not have a header with column names.

- Look at the ".names" text file in the Data Folder, read (visually) the column names and store them in a list
- the url is https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
- read the file with `read_csv` using also the `names` parameter
- show the head of the file, just for a quick inspection

Notice: Iris is also available directly with the datasets package, together with some other *toy* datasets used as examples

[9]:
```
names = ['sep-len', 'sep-wid', 'pet-len', 'pet-wid', 'class']
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
df = pd.read_csv(url, sep = ',', names = names)
df.head()
```

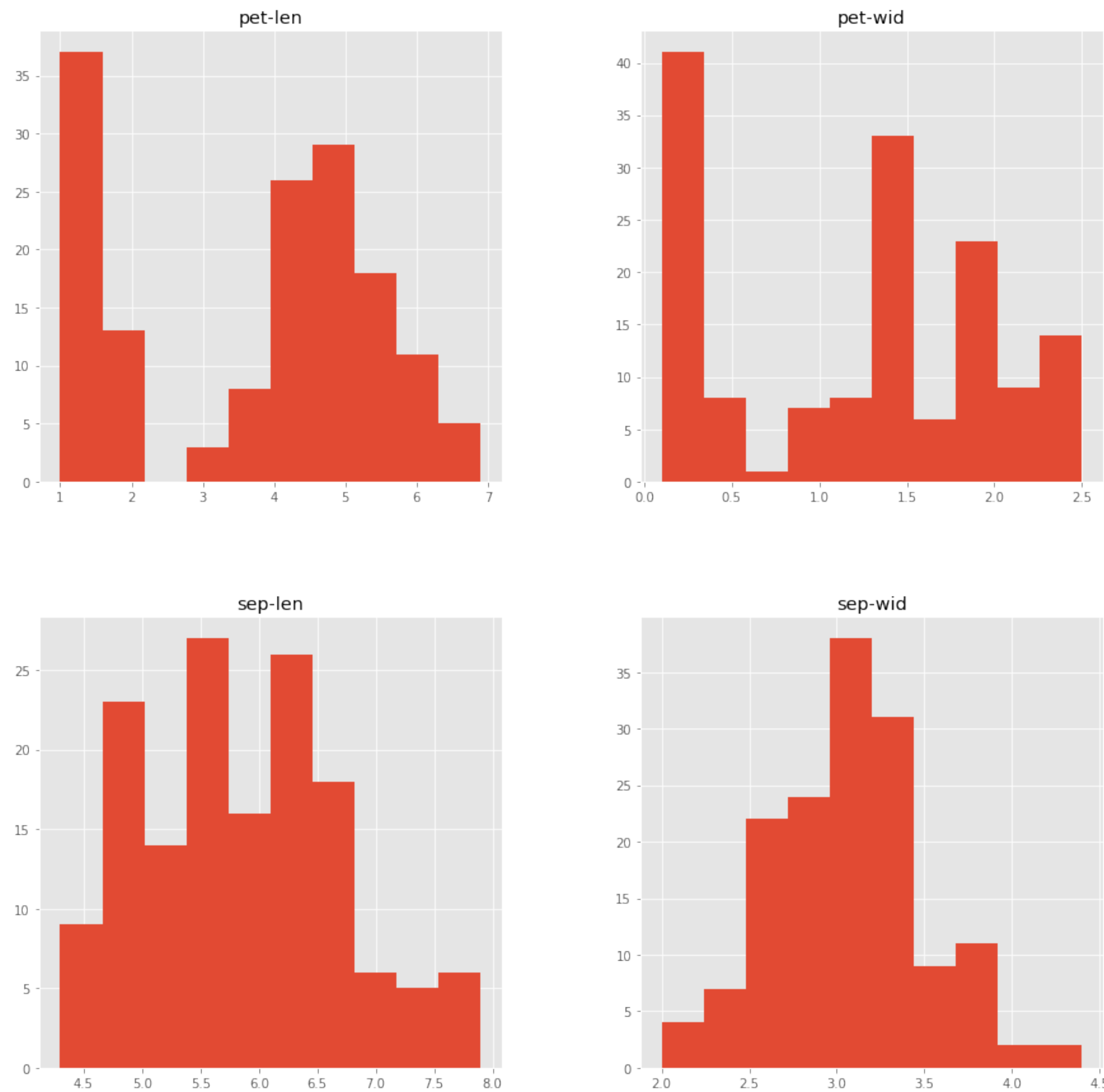
[9]:

	sep-len	sep-wid	pet-len	pet-wid	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

2 Print histogram of numeric values

Use the `hist` method again

[10]:
```
pd.DataFrame.hist(df, figsize = [15,15]);
```

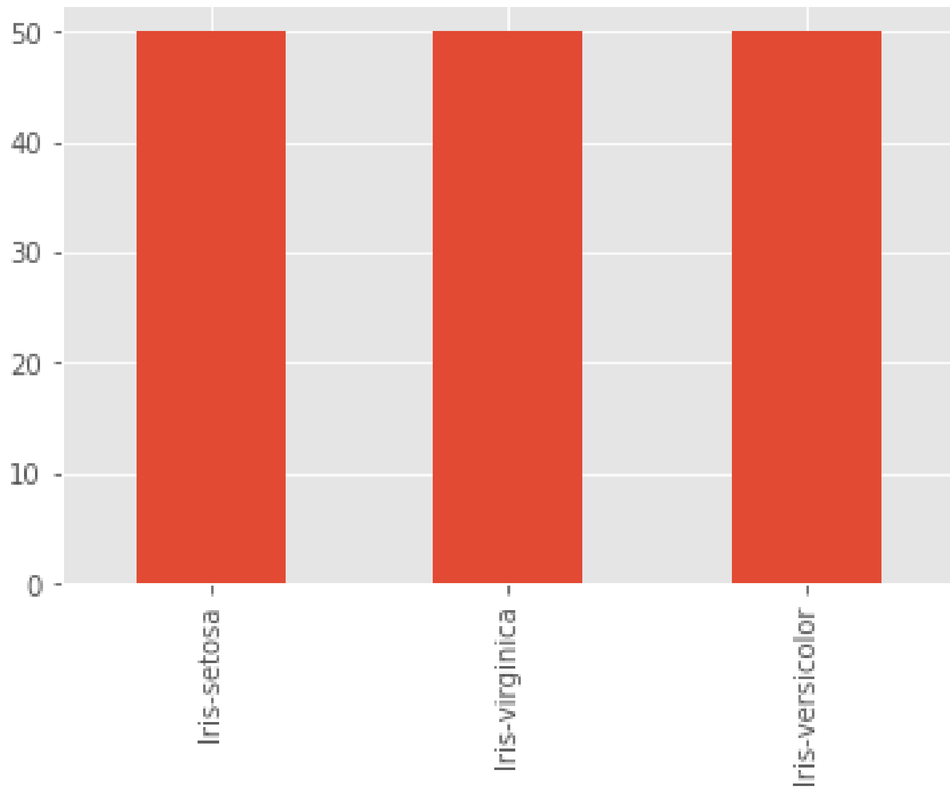


2.0.1 Print histogram of frequencies for the class value

Use the `value_count` method on `class` column, then plot the result with `kind = 'bar'`

[11]:
```
df['class'].value_counts().plot(kind = 'bar')
```

[11]: <matplotlib.axes._subplots.AxesSubplot at 0x228c9c15da0>


50
40
30
20
10
0
Iris-setosa
Iris-virginica
Iris-versicolor

dataMining02-data_exploration-wines

February 12, 2020

1 From UCI Machine Learning Repository

[1]:
```
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
plt.style.use('ggplot')
```

1.1 Wine Quality Dataset

1.1.1 Read data from archive.

In this case, it is a csv with header In this case, it is a csv with header, separator is ';' The download url is http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv

Use the read_csv() method of pandas dataframe https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

Use `df` as the dataframe name

In this dataset the column names are already included in the .csv file

[2]:
```
url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/
 ↪winequality-red.csv'
df = pd.read_csv(url, sep = ';')
```

1.1.2 Show column names

Use the `columns` attribute of pandas on `df`

[3]:
```
df.columns
```

[3]:
```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'quality'],
      dtype='object')
```

1.1.3 Show portion of data

Use the `head` method of pandas dataframe

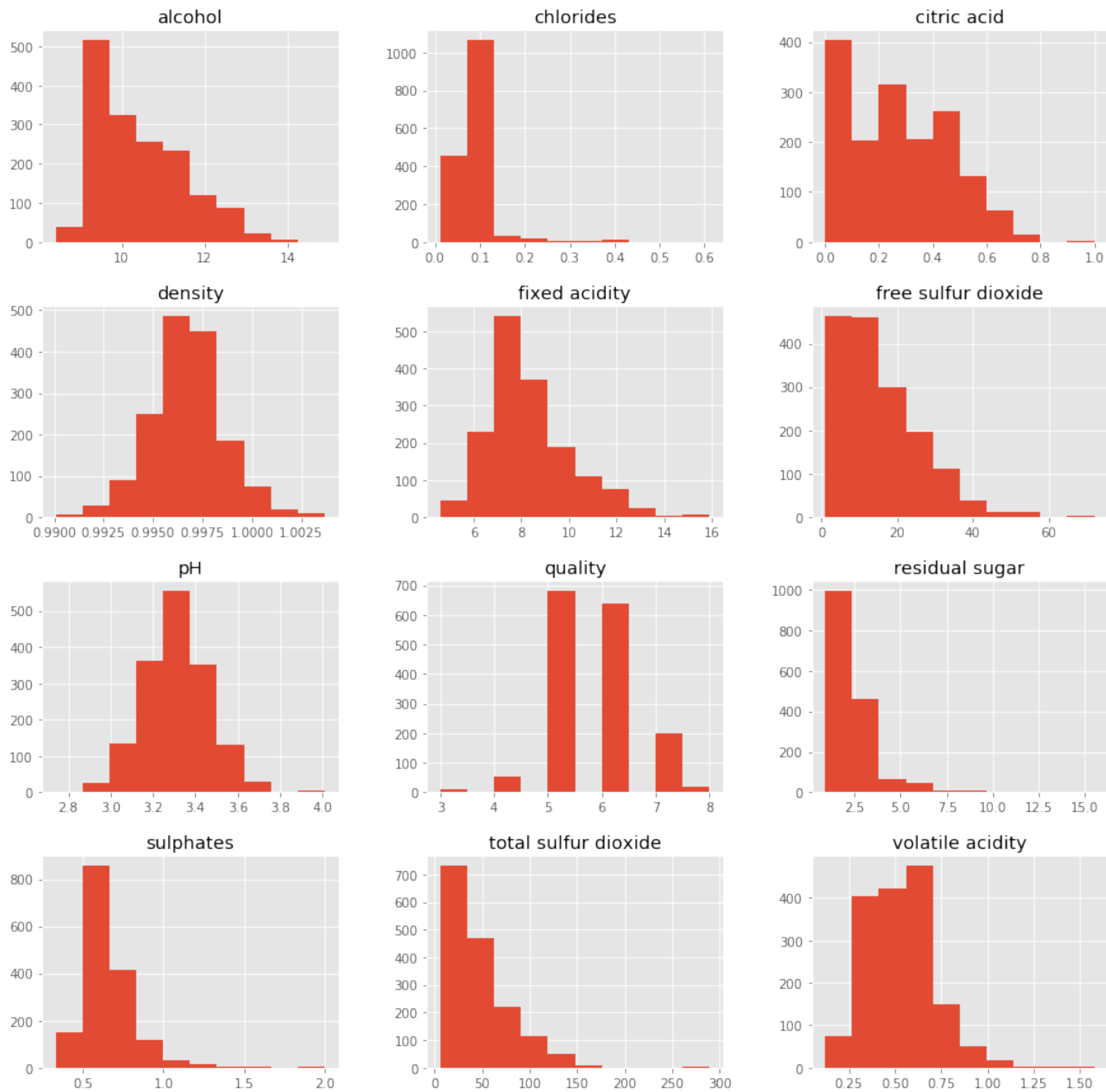
[4]:
```
df.head()
```

[4]:
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality
0      9.4        5
1      9.8        5
2      9.8        5
3      9.8        6
4      9.4        5
```

1.1.4 Show histograms for all numeric values

Use the `DataFrame.hist` method of Pandas. You can set the `figsize` parameter to adjust size

[5]:
```
pd.DataFrame.hist(df, figsize = [15,15]);
```



1.1.5 Show synthetic description

Use the `describe` method of Pandas

[6]:
```
df.describe()
```

[6]:
```
       fixed acidity  volatile acidity  citric acid  residual sugar  \
count    1599.000000       1599.000000  1599.000000     1599.000000
mean        8.319637          0.527821     0.270976        2.538806
std         1.741096          0.179060     0.194801        1.409928
min         4.600000          0.120000     0.000000        0.900000
25%         7.100000          0.390000     0.090000        1.900000
50%         7.900000          0.520000     0.260000        2.200000
```

```
75%         9.200000            0.640000     0.420000         2.600000
max        15.900000            1.580000     1.000000        15.500000

         chlorides  free sulfur dioxide  total sulfur dioxide      density  \
count  1599.000000          1599.000000           1599.000000  1599.000000
mean      0.087467            15.874922             46.467792     0.996747
std       0.047065            10.460157             32.895324     0.001887
min       0.012000             1.000000              6.000000     0.990070
25%       0.070000             7.000000             22.000000     0.995600
50%       0.079000            14.000000             38.000000     0.996750
75%       0.090000            21.000000             62.000000     0.997835
max       0.611000            72.000000            289.000000     1.003690

                pH    sulphates      alcohol      quality
count  1599.000000  1599.000000  1599.000000  1599.000000
mean      3.311113     0.658149    10.422983     5.636023
std       0.154386     0.169507     1.065668     0.807569
min       2.740000     0.330000     8.400000     3.000000
25%       3.210000     0.550000     9.500000     5.000000
50%       3.310000     0.620000    10.200000     6.000000
75%       3.400000     0.730000    11.100000     6.000000
max       4.010000     2.000000    14.900000     8.000000
```

Quality is the target class in this dataset. The **describe** method of pandas dataframes gives a short summary

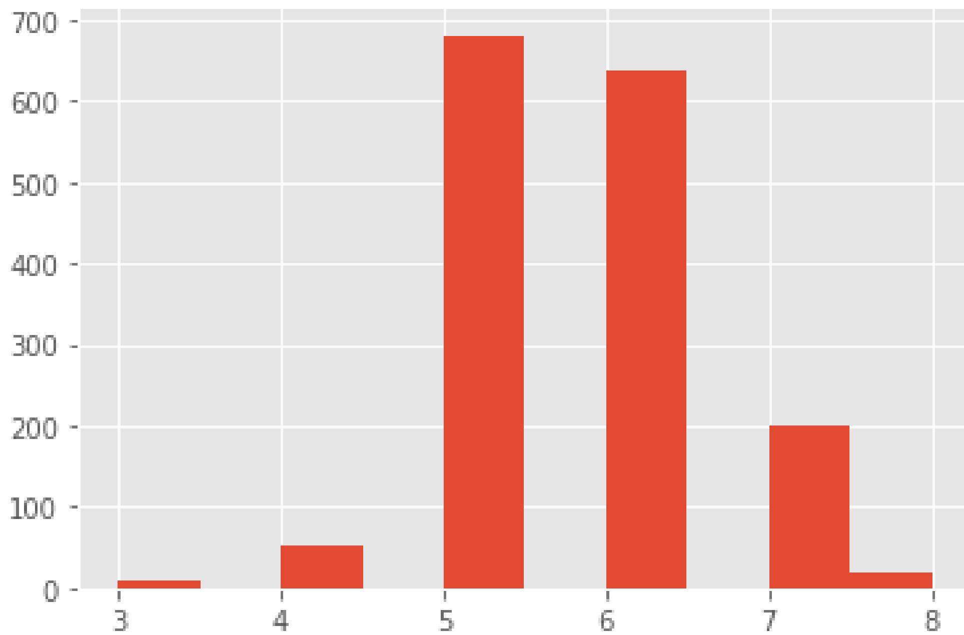
[7]:
```
df['quality'].describe()
```

[7]:
```
count    1599.000000
mean        5.636023
std         0.807569
min         3.000000
25%         5.000000
50%         6.000000
75%         6.000000
max         8.000000
Name: quality, dtype: float64
```

1.1.6 Plot an histogram for "quality"

Use the `hist` method of `matplotlib.pyplot` applied to the `quality` column of `df`

[8]:
```
plt.hist(df['quality'])
plt.show()
```



700
600
500
400
300
200
100
0
3
4
5
6
7
8

ml_03-01-intro-iris

February 12, 2020

This notebook contains an excerpt from the Python Data Science Handbook by Jake VanderPlas; the content is available on GitHub.

Adapded for class presentation by Claudio Sartori - University of Bologna

1 Introducing Scikit-Learn

Scikit-Learn - package that provides efficient versions of a large number of common algorithms - clean, uniform, and streamlined API - very useful and complete online documentation. - once you understand the basic use and syntax of Scikit-Learn for one type of model, switching to a new model or algorithm is very straightforward

1.1 Contents

- *Introduction* to Scikit-Learn
- *Data representation* in Scikit-Learn
- *Estimator* API
- *Examples*

1.2 Data Representation in Scikit-Learn

1.2.1 Data as table

- a two-dimensional grid of data
  - rows represent individual elements of the dataset
  - columns represent quantities related to each of these elements
- Example: Iris dataset
  - analyzed by Ronald Fisher in 1936
  - download this dataset in the form of a Pandas `DataFrame` using the seaborn library

[1]:
```
import seaborn as sns
import pandas as pd
```

Download the **Iris** dataset at the url `https://archive.ics.uci.edu/ml/machine-learning-databases/iris/ir` or from your local file, if you already have it. The file does not have header, use as column names the list below, inspect the text file to see which character is used as separator.

`'sepal length', 'sepal width', 'petal length', 'petal width', 'species'`

Use the dataframe name `iris`. Show the head of `iris`

--> Insert your code in new cell below

[2]:
```
iris_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.
↪data'
iris = pd.read_csv(iris_url, sep = ',', header = None\
                   , names = ['sepal length', 'sepal width', 'petal length',␣
↪'petal width', 'species'])
iris.head(4) # show first 4 data rows
```

[2]:

	sepal length	sepal width	petal length	petal width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa

- each row refers to a single observed flower
  - the number of rows is the total number of flowers in the dataset.
  - *sample*: a single row
  - `n_samples`: number of rows
- each column refers to a piece of information that describes each sample
  - *feature*: a single column `n_features`: the number of columns
    - each column has a data type: number (continuous), boolean, discrete (nominal or ordinal, represented with integers or strings)

Features matrix The part of the data matrix containing the `unsupervised attributes`

Usually in *scikit-learn* documentation referred as `X`

Can be a: - two-dimensional numpy array with shape `[n_samples, n_features]` - SciPy `sparse matrix` - Pandas `DataFrame`

The matrix cases require uniform data types in columns

Target array *label* or *target* array, by convention usually called `y` - usually one dimensional, with length `n_samples`, - generally contained in a NumPy array or Pandas `Series`. - may have continuous numerical values, or discrete classes/labels - usually it the quantity we want to *predict from the data* - in statistical terms, it is the dependent variable

In the example we may wish to construct a model that can predict the species of flower based on the other measurements

The measurements of the flower components are the `features array`

The `species` column can be considered the target array

1.2.2 Visualization

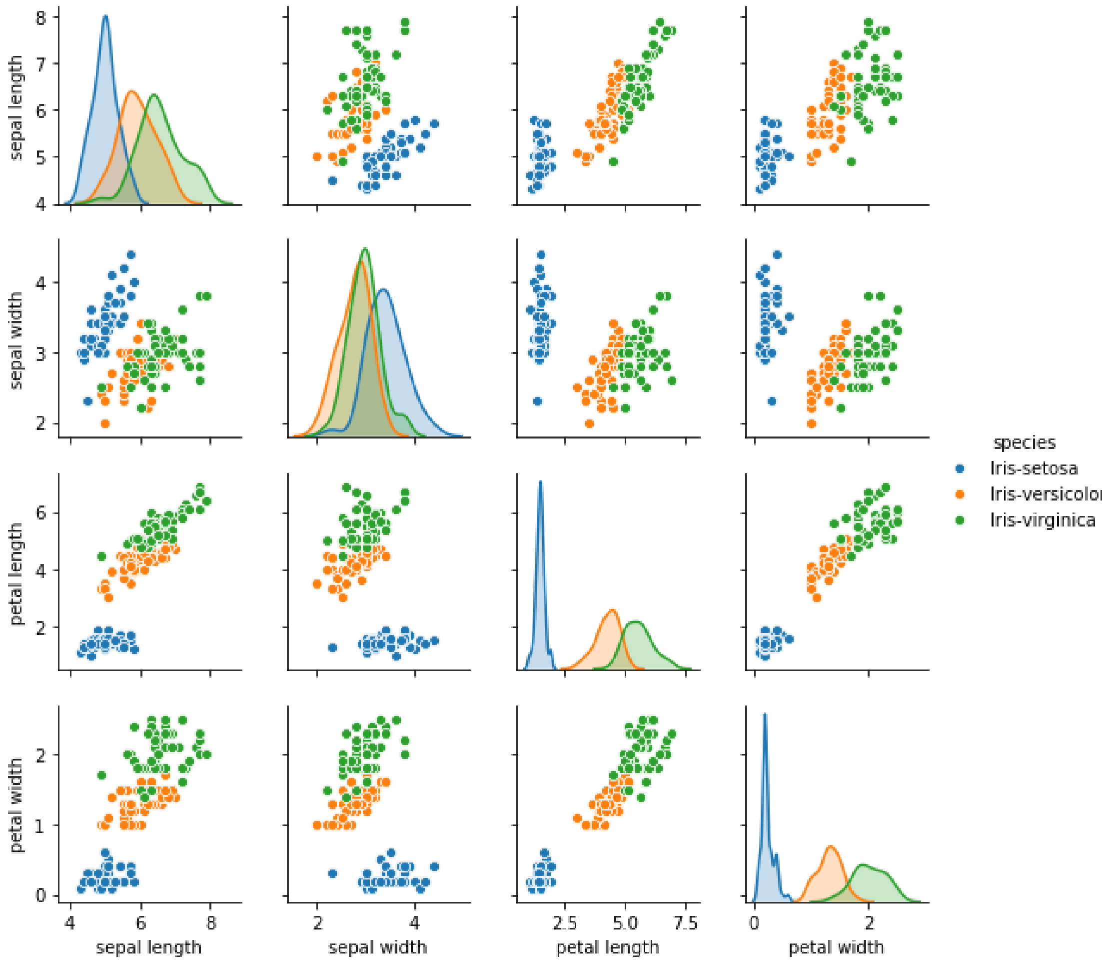
Use Seaborn (see Visualization With Seaborn) to visualize the data

Below we need to prepare the environment for plotting information on the dataset.

1. issue the command `%matplotlib inline` In this way, the output of plotting commands is displayed inline within frontends like the Jupyter notebook, directly below the code cell that produced it. The resulting plots will then also be stored in the notebook document.
2. import `seaborn` giving it the 'nickname' `sns`
3. call the `pairplot` function of `seaborn` on the `iris` dataset, with parameters
   - `hue = 'species'`, this sets the meaning of the color in the plot of the points of the dataset
   - `height = 2`, this sets the size of the plots

--> insert your code in a new cell below this one

[3]:
```
%matplotlib inline
import seaborn as sns
sns.pairplot(iris, hue='species', height=2);
```



For use in Scikit-Learn, we will extract the features matrix and target array from the `DataFrame`. We can do this using some of the Pandas `DataFrame` operations discussed in the Chapter 3 of the above mentioned book.

For example, the `.drop` method allows to drop a column or row by name; remember to specify the axis to use, which is 1 for columns.

1.2.3 Preparing features and target

Store in X the content of `iris` excluding the column `species`. Verify the shape

--> insert your code in a new cell below this one

[4]:
```
X = iris.drop('species', axis=1)
X.shape
```

[4]:
```
(150, 4)
```

Store in `y` the column `species` of `iris`. Verify the shape

--> insert your code in a new cell below this one

[5]:
```
y = iris['species']
y.shape
```

[5]:
```
(150,)
```

1.3 Scikit-Learn's Estimator API

The Scikit-Learn API is designed with the following guiding principles in mind, as outlined in the Scikit-Learn API paper:

- *Consistency*: All objects share a common interface drawn from a limited set of methods, with consistent documentation.
- *Inspection*: All specified parameter values are exposed as public attributes.
- *Limited object hierarchy*: Only algorithms are represented by Python classes; datasets are represented in standard formats (NumPy arrays, Pandas `DataFrame`s, SciPy sparse matrices) and parameter names use standard Python strings.
- *Composition*: Many machine learning tasks can be expressed as sequences of more fundamental algorithms, and Scikit-Learn makes use of this wherever possible.
- *Sensible defaults*: When models require user-specified parameters, the library defines an appropriate default value.

In practice, these principles make Scikit-Learn very easy to use, once the basic principles are understood. Every machine learning algorithm in Scikit-Learn is implemented via the Estimator API, which provides a consistent interface for a wide range of machine learning applications.

1.4 Hyperparameters

The machine learning algorithms are designed to learn from the data the *parameters* that will be used at run time by the algorithms implementing the tasks to perform at the best on data similar to those used in learning.

For example, a *decision tree* (and in particular all the tests placed in the nodes) are the parameters of a *decision tree classifier*

The learning process is also controlled by other parameters (e.g. to control the *overfitting*) which cannot be direcly learned from the data, but are chosen *before* the learning process. Those are called **hyperparameters**

1.4.1 Basics of the API

Most commonly, the steps in using the Scikit-Learn estimator API are as follows (we will step through a handful of detailed examples in the sections that follow).

1. Choose a class of model by importing the appropriate estimator class from Scikit-Learn.
2. Choose model hyperparameters by instantiating this class with desired values.
   - or in the first attempt use the default values
3. Arrange data into a features matrix and target vector following the discussion above.
4. Fit the model to your data by calling the `fit()` method of the model instance.
5. Apply the Model to new data:

- For supervised learning, often we predict labels for unknown data using the `predict()` method.
- For unsupervised learning, we often transform or infer properties of the data using the `transform()` or `predict()` method.

We will now step through several simple examples of applying supervised and unsupervised learning methods.

1.4.2 Supervised learning example: Iris classification

Let's take a look at another example of this process, using the Iris dataset we discussed earlier. Our question will be this: given a model trained on a portion of the Iris data, how well can we predict the remaining labels?

For this task, we will use the *Decision Tree* algorithm, with the standard parameter values. We would like to evaluate the model on data it has not seen before, and so we will split the data into a *training set* and a *testing set*. This could be done by hand, but it is more convenient to use the `train_test_split` utility function

1. Import the method `train_test_split` from `sklearn.model_selection`
2. Generate the variables `Xtrain, Xtest, ytrain, ytest` by calling the function `train_test_split` with parameters `X` and `y`, and the additional parameter `random_state = 1`
3. Show the shape of the resulting variables

--> insert your code in a new cell below this one

[6]:
```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=1)
print(Xtrain.shape, Xtest.shape, ytrain.shape, ytest.shape)
```

```
(112, 4) (38, 4) (112,) (38,)
```

With the data arranged, we can follow our recipe to predict the labels: 1. choose the model class, it will be `DecisionTreeClassifier`, imported from `sklearn.tree` 2. instantiate the `model` as a `DecisionTreeClassifier` whithout any hyperparameter, we will use the defaults 3. fit the `model` to data, calling its method `fit` with parameters `Xtrain, ytrain` 4. predict the target `ytrain_model` using the `predict` method of `model` on the `Xtrain` data

--> insert your code in a new cell below this one

[7]:
```
from sklearn.tree import DecisionTreeClassifier       # 1. choose model class
model = DecisionTreeClassifier(criterion = 'entropy') # 2. instantiate model
model.fit(Xtrain, ytrain)                             # 3. fit model to data
```

```
ytrain_model = model.predict(Xtrain)                    # 4. fir model to␣
 ↪training data
```

We can use the `accuracy_score` utility to see the fraction of predicted training set labels that match their true value.

Import the `accuracy_score` from `sklearn.metrics` and call it on `ytrain, ytrain_model`

--> insert your code in a new cell below this one

[8]:
```
from sklearn.metrics import accuracy_score
accuracy_train = accuracy_score(ytrain, ytrain_model)
print("The accuracy on training set is {0:.2f}%".format(accuracy_train * 100))
```

```
The accuracy on training set is 100.00%
```

Finally, predict the new target `ytest_model` using the `predict` method of `model` on the `Xtest` data, then compute the accuracy on the test set

--> insert your code in a new cell below this one

[9]:
```
ytest_model = model.predict(Xtest)                  # 4. predict on new data
accuracy_test = accuracy_score(ytest, ytest_model)
print("The accuracy on test set is {0:.2f}%".format(accuracy_test * 100))
```

```
The accuracy on test set is 97.37%
```

1.5 Show the Decision Tree

To show the Decision Tree we will need a few imports

`from matplotlib import pyplot` `from sklearn.tree import plot_tre` `from matplotlib.pyplot import figure`

We will start setting the *figure size* with the `figure` function, taking as argument `figsize` and a list of two values in inches, try and error for the measures you like.

We will then use the `plot_tree` function of `sklearn.tree`. It takes as argument the *fitted model*, in our case `model` and several arguments to control how the tree is displayed.

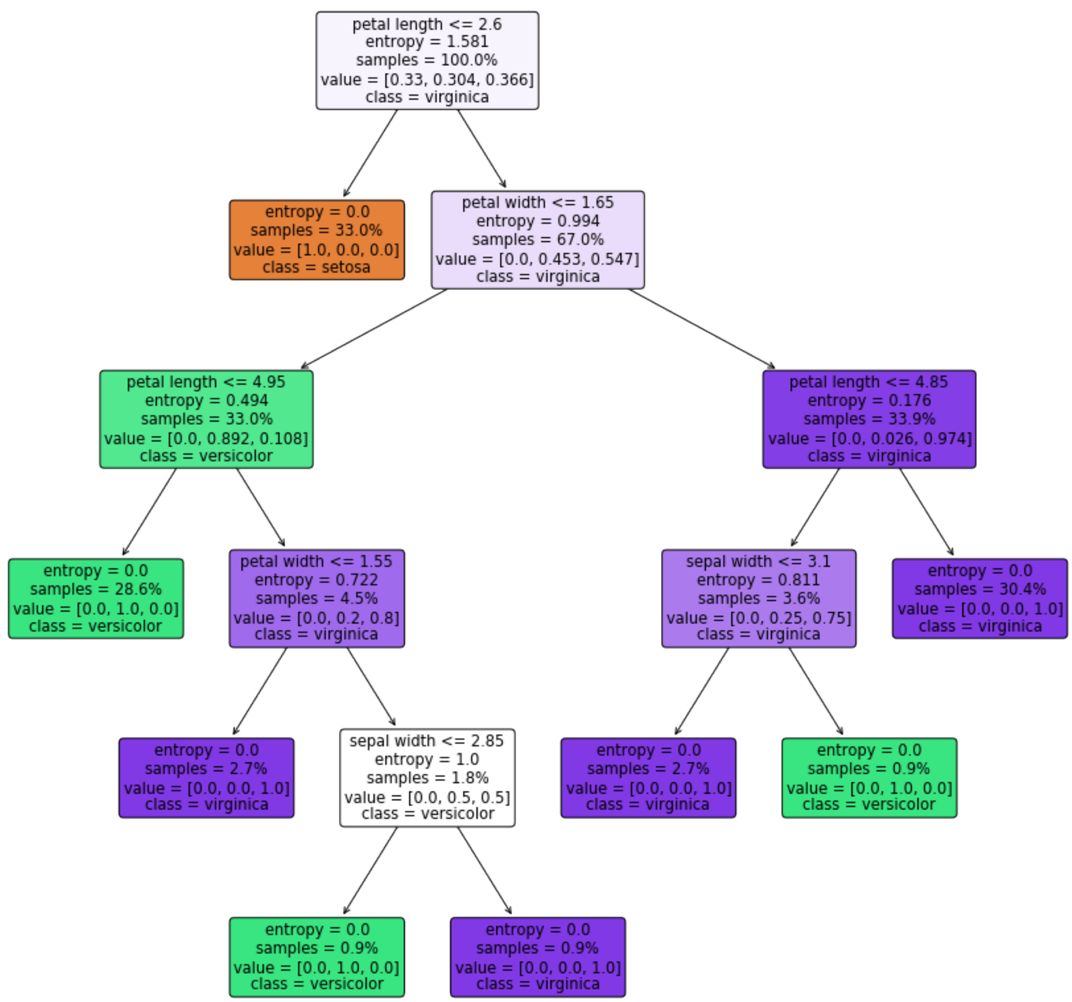
I suggest the arguments below, you can try freely configurations and omissions of the parameters, to use the defaults. The parameters must follow the model variable and be separated by commas, the order is not relevant, since the parameters are named.

`filled=True` `feature_names = ['sepal length', 'sepal width'` , 'petal length', 'petal width']`<br>`class_names = ['setosa', 'versicolor', 'virginica']`<br>`rounded = True`<br>`proportion = True`<br>`rotate = False“

--> insert your code in a new cell below this one

[10]:
```
from matplotlib import pyplot
from sklearn.tree import plot_tree
```

```python
from matplotlib.pyplot import figure
figure(figsize = (15,15))
plot_tree(model
#           , fontsize=6
           , filled=True
           , feature_names = ['sepal length', 'sepal width', 'petal length', 'petal width']
           , class_names = ['setosa', 'versicolor', 'virginica']
           , rounded = True
           , proportion = True
           , rotate = False
          );
```



ml-03-02a-pruning_example

February 12, 2020

1 Pruning the Decision Tree

In this example we are directly given two different datasets, one will be used for training, the other for testing.

We will start training the model with the training data, then testing it with the test data.

Then we will observe the resulting tree, and try to improve the result with pruning.

Start importing `pandas` and `numpy`, then assign to the variables `in_train` and `in_test` the strings `binaries_train.csv` and `binaries_test.csv`.

--> Insert your code in a new cell after this one

[1]:
```
import pandas as pd
import numpy as np
in_train = 'binaries_train.csv'
in_test = 'binaries_test.csv'
```

Read the `in_train` file into the `train` dataframe and inspect it.

--> Insert your code in a new cell after this one

[2]:
```
train = pd.read_csv(in_train)
train.head()
```

[2]:

	B1	B2	B3	B4	B5	B6	Class
0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1
2	0	0	0	0	1	0	0
3	0	0	0	0	1	1	1
4	0	0	0	1	0	0	1

Prepare the `X_train` variable, by dropping the last column of `train`, and the `y_train` variable with the last column of `train`, then inspect the shapes.

--> Insert your code in a new cell after this one

[3]:
```
X_train = train.drop(train.columns[-1], axis = 1) # drop the last column
X_train.shape
```

[3]: (64, 6)

[4]:
```
#y_train = train.drop(train.columns[0:-1], axis = 1) # drop all the columns but␣
↪the last
y_train = train.iloc[:,-1]
y_train.shape
```

[4]: (64,)

1.1 Train a full tree

With the `fit` method we will train the **Decision Tree**.

The parameters for the training are set in the creation of the `model`.

In this case we will set only the 'split criterion' to `entropy`. Don't forget to import the `tree.DecisionTreeClassifier` from `sklearn`.

--> Insert your code in a new cell after this one

[5]:
```
from sklearn import tree
model = tree.DecisionTreeClassifier(criterion="entropy")
model.fit(X_train, y_train)
```

[5]:
```
DecisionTreeClassifier(class_weight=None, criterion='entropy', max_depth=None,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

Now use the trained `model` to predict `y_predicted_train` from `X_train`.

Evaluate the percentage of matches between `y_predicted_train` and `y_train`.

Hint: you can use `np.mean` on the comparison of the two vectors.

--> Insert your code in a new cell after this one

[6]:
```
y_predicted_train = model.predict(X_train)
```

[7]:
```
accuracy_train = np.mean(y_train == y_predicted_train) * 100
print("The accuracy on training set is {0:.1f}%".format(accuracy_train))
```

```
The accuracy on training set is 100.0%
```

Now we load the test set, make the prediction using the already trained model and compute the accuracy.

--> Insert your code in a new cell after this one

[8]:
```
test=pd.read_csv(in_test)
X_test = test.drop(test.columns[-1], axis = 1) # drop the last column
y_test = test.iloc[:,-1] # keep only the last column
y_predicted_test = model.predict(X_test)
accuracy_test = np.mean(y_test == y_predicted_test) * 100
print("The accuracy on test set is {0:.1f}%".format(accuracy_test))
```

```
The accuracy on test set is 60.9%
```

1.1.1 Observe the tree

Import the following and set the size of the figure as below

`from matplotlib.pyplot import figure figure(figsize = (25,25))`

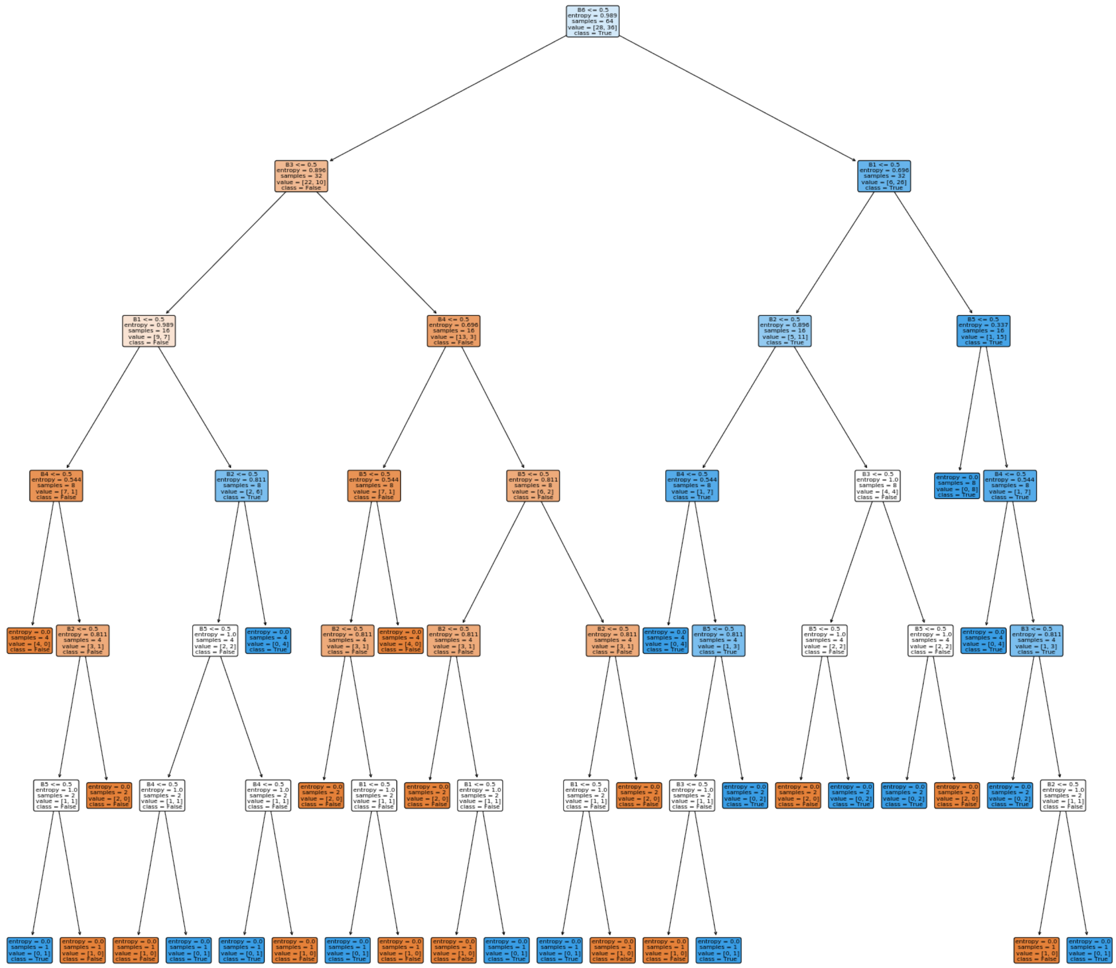
Plot the tree with the `tree.plot_tree` function. Use as parameters the `model, rounded = True, filled = True`.

Use as `feature_names` the column names of train. Use as `class_names` 'False' and 'True'

The tree generated has a number of internal node levels equal to the number of predicting attributes.

--> Insert your code in a new cell after this one

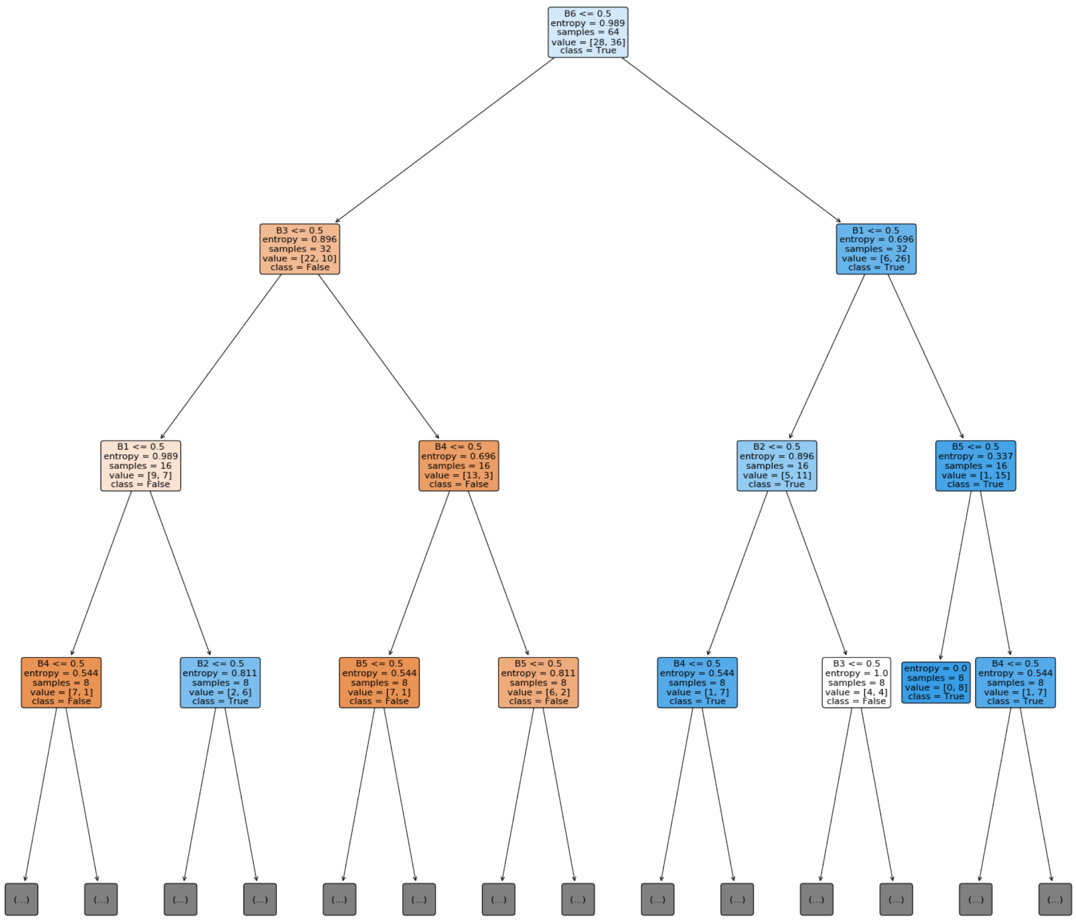
[13]:
```
from matplotlib.pyplot import figure
figure(figsize = (25,25))
tree.plot_tree(model
           , filled=True
           , feature_names = train.columns
           , class_names = ['False', 'True']
           , rounded = True
          );
```



Try to understand better the tree by plotting only the first two levels under the root. This is obtained with the parameter `max_depth = 3`. Remember that here we are not changing the tree, but only displaying the upper part of the tree.

--> Insert your code in a new cell after this one

[10]:
```
figure(figsize = (25,25))
tree.plot_tree(model
          , filled=True
          , feature_names = train.columns
          , class_names = ['False', 'True']
          , rounded = True
          , max_depth = 3
         );
```



1.2 Pruned tree

From the observation of the tree, choose an appropriate value for `max_dept` and redo the training using the parameter `max_depth = max_depth` in the `fit` method. Compute the accuracy on the training set, and then on the test set.

--> Insert your code in a new cell after this one

[11]:
```
max_depth = 1
model_pruned = tree.DecisionTreeClassifier(criterion="entropy", max_depth =␣
↪max_depth)
model_pruned.fit(X_train, y_train)
y_predicted_train_pruned = model_pruned.predict(X_train)
```

```
accuracy_train_pruned = np.mean(y_train == y_predicted_train_pruned) * 100
print("The accuracy of the pruned tree on training set is {0:.1f}%".
 ↪format(accuracy_train_pruned))
```

```
The accuracy of the pruned tree on training set is 75.0%
```

[12]:
```
y_predicted_test_pruned = model_pruned.predict(X_test)
accuracy_test_pruned = np.mean(y_test == y_predicted_test_pruned) * 100
print("The accuracy of the pruned tree on test set is {0:.1f}%".
 ↪format(accuracy_test_pruned))
```

```
The accuracy of the pruned tree on test set is 76.6%
```

ml-03-02b-class-w-hyperp-tuning

February 12, 2020

1 Classification with hyperparameter tuning

1.0.1 aim:

Show classification with different strategies for the tuning and evaluation of the classifier 1. simple **holdout** 2. **holdout with validation** train and validate repeatedly changing a hyperparameter, to find the value giving the best score, then test for the final score 3. **cross validation** on training set, then score on test set 4. **bagging** it is an *ensemble* method made available in `scikit-learn`

NB: You should not interpret those experiments as a way to find the *best* evaluation method, but simply as examples of *how* to do the evaluation.

If you look at the final report, methods **1** to **3** are meant for increasing evaluation reliability, method **3** is the more reliable, but it requires several repetitions for cross validation, therefore, if the learning method is expensive, it requires long processing time. If, due to intrinsic variation caused by random sampling, it turns out that methods **1** or **2** give higher accuracy, this means simply that the forecast towards generalisation is less reliable.

Method **4** is on a different dimension, it simply shows that a good result can be obtained with an *ensemble* of simpler classifiers (the best value for the hyperparameter `max_depth` is smaller than in the other cases)

1.0.2 Workflow

- download the data
- drop the useless data
- separe the predicting attributes X from the class attribute y
- split X and y into training and test

- part 1 - single run with default parameters
  - initialise an estimator with the chosen model generator
  - fit the estimator with the training part of X
  - show the tree structure
  - part 1.1
    - predict the y values with the fitted estimator and the train data

            - compare the predicted values with the true ones and compute the accuracy on the training set
  - part 1.2
    - predict the y values with the fitted estimator and the test data
      - compare the predicted values with the true ones and compute the accuracy on the test set

- part 2 - multiple runs changing a parameter
  - prepare the structure to hold the accuracy data for the multiple runs
  - repeat for all the values of the parameter
    - initialise an estimator with the current parameter value
    - fit the estimator with the training part
    - predict the class for the test part
    - compute the accuracy and store the value
  - find the parameter value for the top accuracy

- part 3 - compute accuracy with cross validation
  - prepare the structure to hold the accuracy data for the multiple runs
  - repeat for all the values of the parameter
    - initialise an estimator with the current parameter value
    - compute the accuracy with cross validation and store the value
  - find the parameter value for the top accuracy
  - fit the estimator with the entire X
  - show the resulting tree and classification report

The data are already in your folder, use the name `winequality-red.csv`

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import tree
from sklearn.metrics import accuracy_score, classification_report,␣
↪confusion_matrix
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import BaggingClassifier

%matplotlib inline
plt.rcParams['figure.figsize'] = [20, 20]
random_state = 15
np.random.seed(random_state)
# the random state is reset here in numpy, all the scikit-learn procedure use␣
↪the numpy random state
# obviously the experiment can be repeated exactly only with a complete run of␣
↪the program

#data_url = "uci_breast_tissue_data/BreastTissue.csv"
data_url = "winequality-red.csv"
target_name = 'quality'
to_drop = []
```

```
# parameter_values to be determined after the fitting of the full tree
df = pd.read_csv(data_url , sep = ';')
print("Shape of the input data {}".format(df.shape))
```

```
Shape of the input data (1599, 12)
```

Have a quick look to the data. - use the .shape attribute to see the size - use the `.head()` function to see column names and some data - use the `.hist()` method for an histogram of the columns - use the .unique method to see the class values

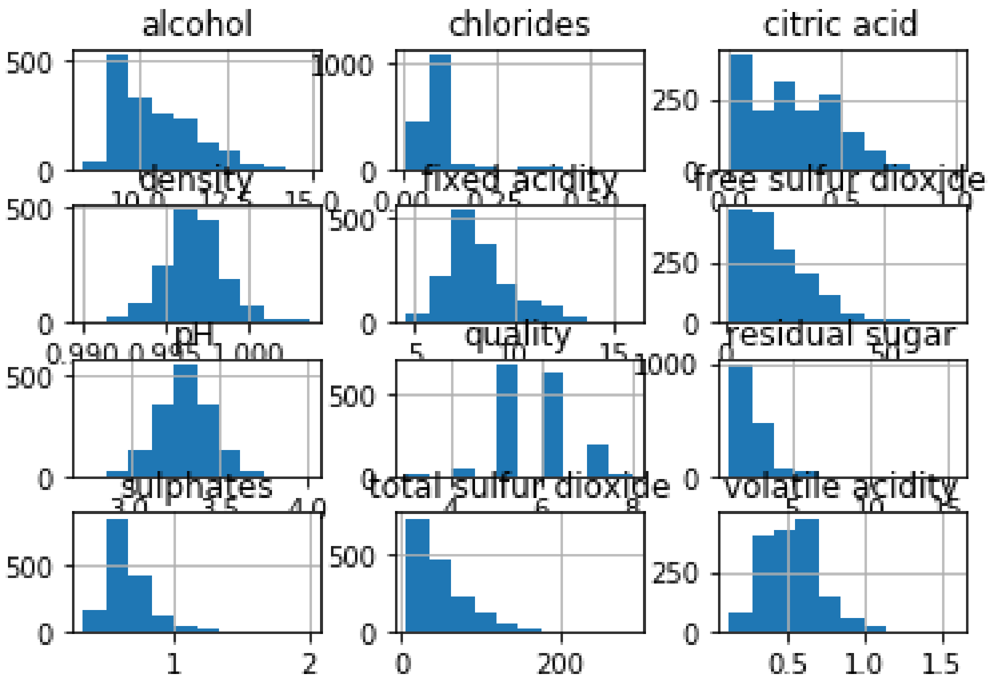
[2]:
```
df.head()
```

[2]:
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality
0      9.4        5
1      9.8        5
2      9.8        5
3      9.8        6
4      9.4        5
```

Use the `hist` method of the DataFrame to show the histograms of the attributes

NB: a semicolon at the end of a statement suppresses the `Out[]`

[3]:
```
df.hist();
```



Print the unique class labels (hint: use the `unique` method of pandas Series

[4]:
```
classes = df[target_name].unique()
classes.sort()
print(classes)
```

```
[3 4 5 6 7 8]
```

Split the data into the predicting values X and the class y Drop also the columns which are not relevant for training a classifier, if any

The method "drop" of dataframes allows to drop either rows or columns - the "axis" parameter chooses between dropping rows (axis=0) or columns (axis=1)

[5]:
```
X = df.drop([target_name], axis = 1) # drop the class column
y = df[target_name] # Class only
```

Another quick look to data

[6]:
```
X.head()
```

[6]:
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
```

```
3          11.2              0.28         0.56             1.9      0.075
4           7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol
0      9.4
1      9.8
2      9.8
3      9.8
4      9.4
```

[7]:
```
y.head()
```

[7]:
```
0    5
1    5
2    5
3    6
4    5
Name: quality, dtype: int64
```

1.1 Prepare a simple model selection: holdout method

- Split X and y in train and test
- Show the number of samples in train and test, show the number of features

[8]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state =␣
 ↪random_state) # default Train 0.75- Test 0.25
print("There are {} samples in the training dataset".format(X_train.shape[0]))
print("There are {} samples in the testing dataset".format(X_test.shape[0]))
print("Each sample has {} features".format(X_train.shape[1]))
```

```
There are 1199 samples in the training dataset
There are 400 samples in the testing dataset
Each sample has 11 features
```

1.2 Part 1

- Initialize an estimator with the required model generator `tree.DecisionTreeClassifier(criterion="entropy")`

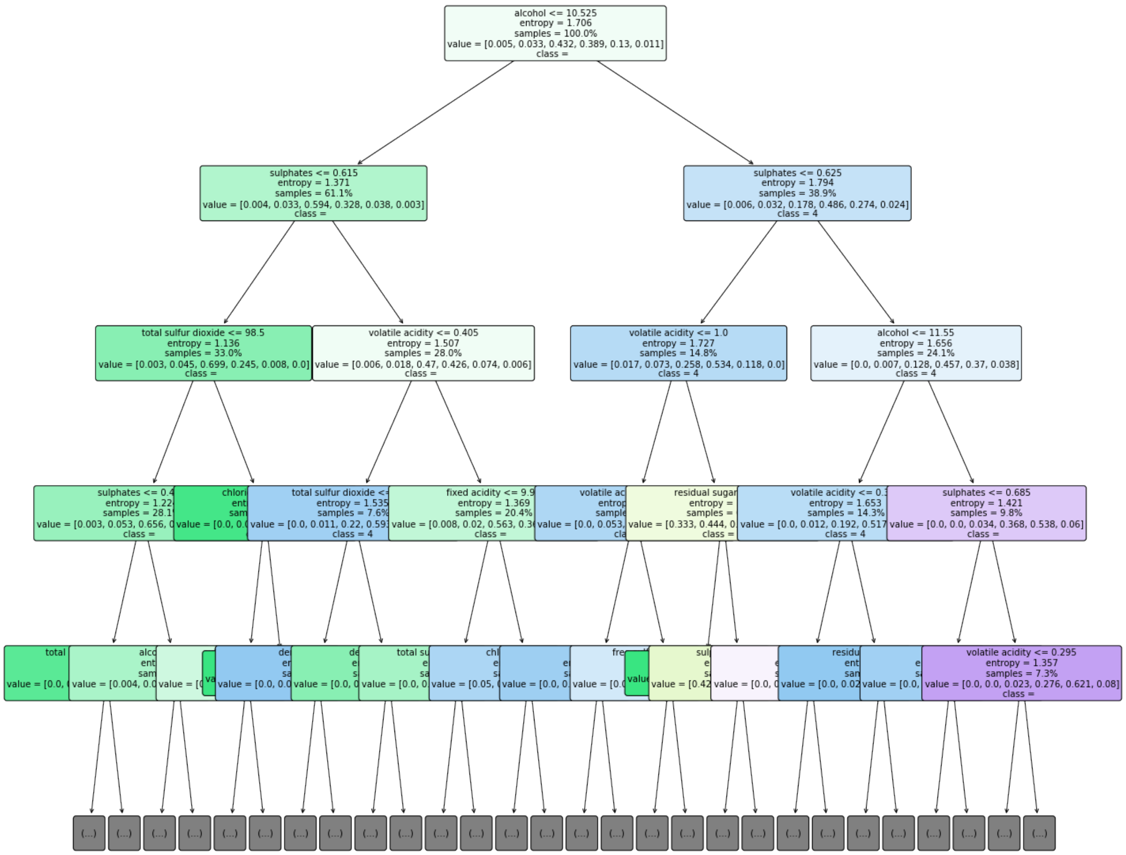
- Fit the estimator on the train data and target

[9]:
```
estimator = tree.DecisionTreeClassifier(criterion="entropy")
estimator.fit(X_train, y_train);
```

Look at the tree structure - the feature names are used to show the tests in the nodes - they are the column names in the X - the class names - the attribute `estimator.classes_` contains the array of classes detected in the target; if the classes are numbers they have to be transformed in strings with `str()` - the dept of the visualization can be limited with the parameter `max_depth`

```
plt.figure(figsize = (20,20))  tree.plot_tree(estimator            , filled=True
, feature_names = X.columns            , class_names = str(estimator.classes_)
, rounded = True            , proportion = True            , max_depth = 1
);
```

[10]:
```
plt.figure(figsize = (20,20),
#           dpi = 500, # this increments the detail, to do a more detiled␣
↪inspection
          )
tree.plot_tree(estimator
          , filled=True
          , feature_names = X.columns
          , class_names = str(estimator.classes_)
          , rounded = True
          , proportion = True
          , fontsize = 10
          , max_depth = 4 # limited view, since the full tree is very complex
              );
```



1.2.1 Part 1.1

Let's see how it works on training data - predict the target using the fitted estimator on the training data - compute the accuracy on the training set using `accuracy_score(<target>,<predicted_target) * 100`

[11]:
```
y_predicted_train = estimator.predict(X_train)
accuracy_train = accuracy_score(y_predicted_train, y_train)*100
print("The accuracy on training set is {0:.1f}%".format(accuracy_train))
```

```
The accuracy on training set is 100.0%
```

1.2.2 Part 1.2

That's more significant: how it works on test data - use the fitted estimator to predict using the test features - compute the accuracy and store it on a variable for the final summary - store the maximum depth of the tree, for later use - `fitted_max_depth = estimator.tree_.max_depth` - store the range of the parameter which will be used for tuning - `parameter_values = range(1,fitted_max_depth+1)` - print the accuracy on the test set and the maximum depth of the tree

[12]:
```
y_predicted_test = estimator.predict(X_test)
accuracy_ho = accuracy_score(y_test, y_predicted_test) * 100
fitted_max_depth = estimator.tree_.max_depth
print("The accuracy on test set is {0:.1f}%".format(accuracy_ho))
print("The maximum depth of the fitted tree is {}".format(fitted_max_depth))
parameter_values = range(1,fitted_max_depth+1)
```

```
The accuracy on test set is 58.8%
The maximum depth of the fitted tree is 16
```

1.3 Part 2

Optimising the tree: limit the maximum tree depth. We will use the three way splitting: `train, validation, test`. For simplicity, since we already splitted in *train* and *test*, we will furtherly split the *train* - split the training set into two parts: **train_t** and **val** - max_depth - pruning the tree cutting the branches which exceed max_depth - the experiment is repeated varying the parameter from 1 to the depth of the unpruned tree - the scores for the various values are collected and plotted

[13]:
```
X_train_t, X_val, y_train_t, y_val = train_test_split(X_train
                                                      , y_train
                                                      , random_state =␣
 ↪random_state) # default Train 0.75- Test 0.25
print("There are {} samples in the training dataset".format(X_train_t.shape[0]))
print("There are {} samples in the validation dataset".format(X_val.shape[0]))
```

```
There are 899 samples in the training dataset
There are 300 samples in the validation dataset
```

1.3.1 Loop for computing the score varying the hyperparameter

- initialise a list to contain the scores
- loop varying `par` in `parameter_values`
  - initialize an estimator with a DecisionTreeClassifier, using `par` as maximum depth and `entropy` as criterion
  - fit the estimator on the `train_t` part of the features and the target
  - predict with the estimator using the validation features
  - compute the score comparing the prediction with the validation target and append it to the end of the list

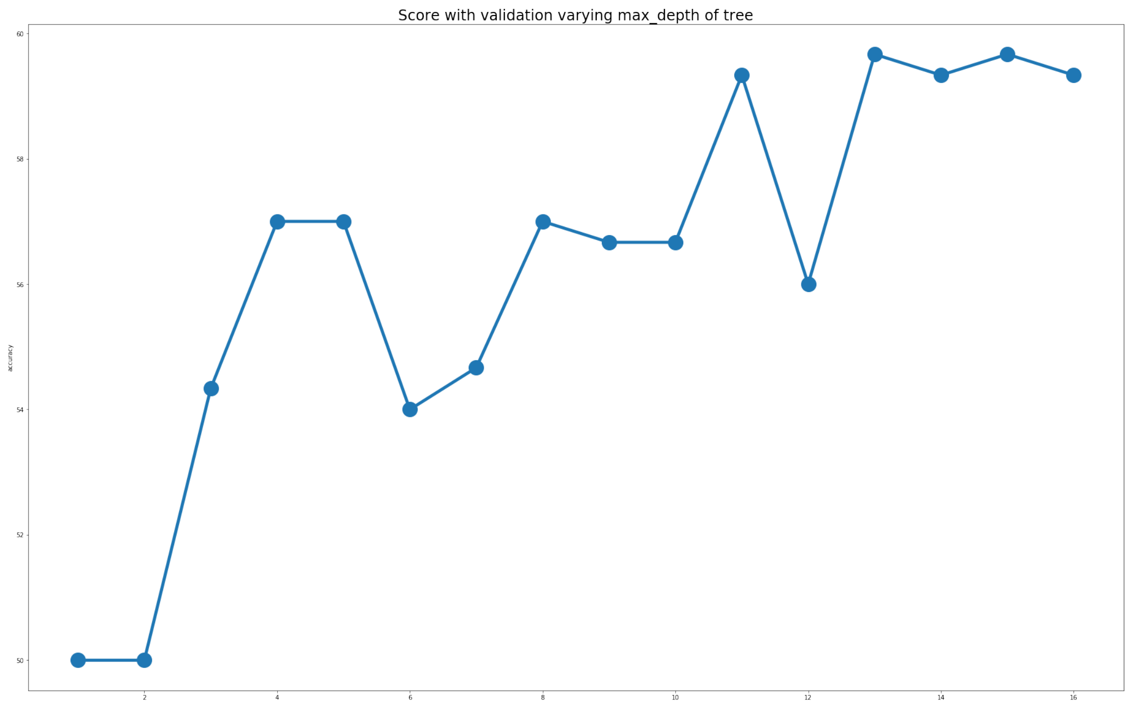
[14]:
```
scores = []
# all_scores = []
```

```python
# parameter_values = [10**exp for exp in np.arange(-2.0,0.0, 0.25)]
#parameter_values = np.arange(0,1, 0.05)
for par in parameter_values:
    estimator = tree.DecisionTreeClassifier(criterion="entropy"
                                             , max_depth = par
                                            )
    estimator.fit(X_train_t, y_train_t)
    y_predicted_val = estimator.predict(X_val)
    score =  accuracy_score(y_val, y_predicted_val) * 100 # compute the matches between prediction and true classes
    scores.append(score)
```

1.3.2 Plot the results

Plot using the `parameter_values` and the list of `scores`

[15]:
```python
plt.figure(figsize=(32,20))
plt.plot(parameter_values, scores, '-o', linewidth=5, markersize=24)
plt.xlabel('max_depth')
plt.ylabel('accuracy')
plt.title("Score with validation varying max_depth of tree", fontsize = 24)
plt.show();
```



1.3.3 Fit the tree after validation and print summary

- store the parameter value giving the best score with `np.argmax(scores)`
- initialize an estimator as a DecisionTreeClassifier, using the best parameter value computed above as maximum depth and `entropy` as criterion
- fit the estimator using the `train` part
- use the fitted estimator to predict using the test features
- compute the accuracy on the test and store it on a variable for the final summary
- print the accuracy on the test set and the best parameter value

[16]:
```
top_par_hov = parameter_values[np.argmax(scores)]
estimator = tree.DecisionTreeClassifier(criterion="entropy", max_depth =␣
↪top_par_hov)
estimator.fit(X_train, y_train)
y_predicted_test = estimator.predict(X_test)
accuracy_hov = accuracy_score(y_predicted_test, y_test) * 100
print("The top accuracy is {0:.1f}%".format(accuracy_hov))
print("Obtained with max_depth = {}".format(top_par_hov))
```

```
The top accuracy is 56.0%
Obtained with max_depth = 13
```

1.4 Part 3 - Tuning with Cross Validation

Optimisation of the hyperparameter with **cross validation** (cv suffix in the variable names). Now we will tune the hyperparameter looping on cross validation with the **training set**, then we will fit the estimator on the training set and evaluate the performance on the **test set**

- initialize an empty list for the scores
- loop varying `par` in `parameter_values`
  - initialize an estimator with a DecisionTreeClassifier, using `par` as maximum depth and `entropy` as criterion
  - compute the score using the estimator on the `train` part of the features and the target using
    - `cross_val_score(estimator, X_train, y_train, scoring='accuracy', cv = 5)`
    - the result is list of scores
  - compute the average of the scores and append it to the end of the list
- print the scores

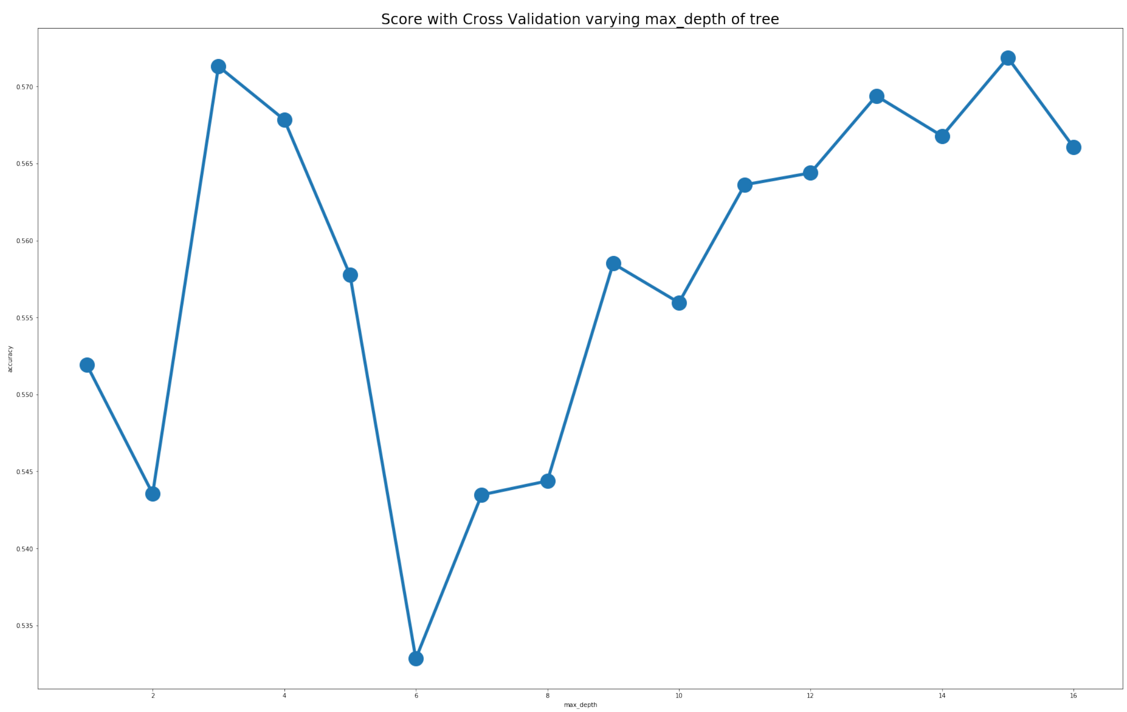
[17]:
```
avg_scores = []
for par in parameter_values:
    estimator = tree.DecisionTreeClassifier(criterion="entropy"
                                            , max_depth = par
                                            )
    scores = cross_val_score(estimator, X_train, y_train
                             , scoring='accuracy', cv = 5)
    # cross_val_score produces an array with one score for each fold
```

```python
    avg_scores.append(np.mean(scores))
print(avg_scores)
```

```
[0.5519152314979172, 0.5435525832020326, 0.5713089389088722, 0.5678437532607689,
0.5577529689800829, 0.5328357011331578, 0.5434774526459061, 0.5443793438898566,
0.5585019217156184, 0.5559627582243493, 0.5636209817495066, 0.5643876543587679,
0.5693922353804821, 0.5667868463793229, 0.571864056684633, 0.566054670861477]
```

Plot using the `parameter_values` and the list of `scores`

[18]:
```python
plt.figure(figsize=(32,20))
plt.plot(parameter_values, avg_scores, '-o', linewidth=5, markersize=24)
plt.xlabel('max_depth')
plt.ylabel('accuracy')
plt.title("Score with Cross Validation varying max_depth of tree", fontsize =␣
 ↪24)
plt.show();
```



1.4.1 Fit the tree after cross validation and print summary

- store the parameter value giving the best score with `np.argmax(scores)`
- initialize an estimator as a DecisionTreeClassifier, using the best parameter value computed above as maximum depth and `entropy` as criterion
- fit the estimator using the `train` part

- use the fitted estimator to predict using the test features
- compute the accuracy on the test and store it on a variable for the final summary
- print the accuracy on the test set and the best parameter value

[19]:
```
top_par_cv = parameter_values[np.argmax(avg_scores)]
estimator = tree.DecisionTreeClassifier(criterion="entropy", max_depth =␣
 ↪top_par_cv)
estimator.fit(X_train,y_train);
y_predicted = estimator.predict(X_test)
accuracy_cv = accuracy_score(y_test, y_predicted) * 100
print("The accuracy on test set tuned with cross_validation is {:.1f}% with␣
 ↪depth {}".format(accuracy_cv, top_par_cv))
```

```
The accuracy on test set tuned with cross_validation is 59.2% with depth 15
```

`print(classification_report(y_test, y_predicted))`

[20]:
```
print(classification_report(y_test, y_predicted))
```

```
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         4
           4       0.00      0.00      0.00        14
           5       0.65      0.69      0.67       163
           6       0.61      0.58      0.59       171
           7       0.46      0.58      0.52        43
           8       0.25      0.20      0.22         5

    accuracy                           0.59       400
   macro avg       0.33      0.34      0.33       400
weighted avg       0.58      0.59      0.59       400
```

- **micro**: Calculate metrics globally by counting the total true positives, false negatives and false positives.
- **macro**: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.
- **weighted**: Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

`print(confusion_matrix(y_test, y_predicted))`

[21]:
```
print(confusion_matrix(y_test, y_predicted))
```

```
[[  0   0   4   0   0   0]
 [  0   0   8   5   1   0]
 [  1   6 112  40   4   0]
 [  0   2  44  99  23   3]
```

```
 [  0   0   3  15  25   0]
 [  0   0   0   3   1   1]]
```

2 4. Tuning with an ensemble method

We will use the **bagging** method, made available by `scikit-learn`, for documentation see the pdf file provided, or the online documentation. - initialize an empty list for the scores - loop varying `par` in `parameter_values` - initialize an estimator with a BaggingClassifier applied to a DecisionTreeClassifier, using `par` as maximum depth and `entropy` as criterion (see below the statement) - fit the estimator on the `train` part - compute an array of cv_scores using the cross validation method on the estimator on the `train` part of the features and the target, with the `accuracy` as scoring and 5 folds, then append the mean of the cv_scores obtained to the end of the list - print the list of scores

```
# use this in the loop for tuning the hyperparameter          # estimator_bagging
= BaggingClassifier(tree.DecisionTreeClassifier(criterion="entropy"
#                                                                     , max_depth
= par   #                                                                   )   #
, max_samples=0.5, max_features=0.5 #                                      )
```

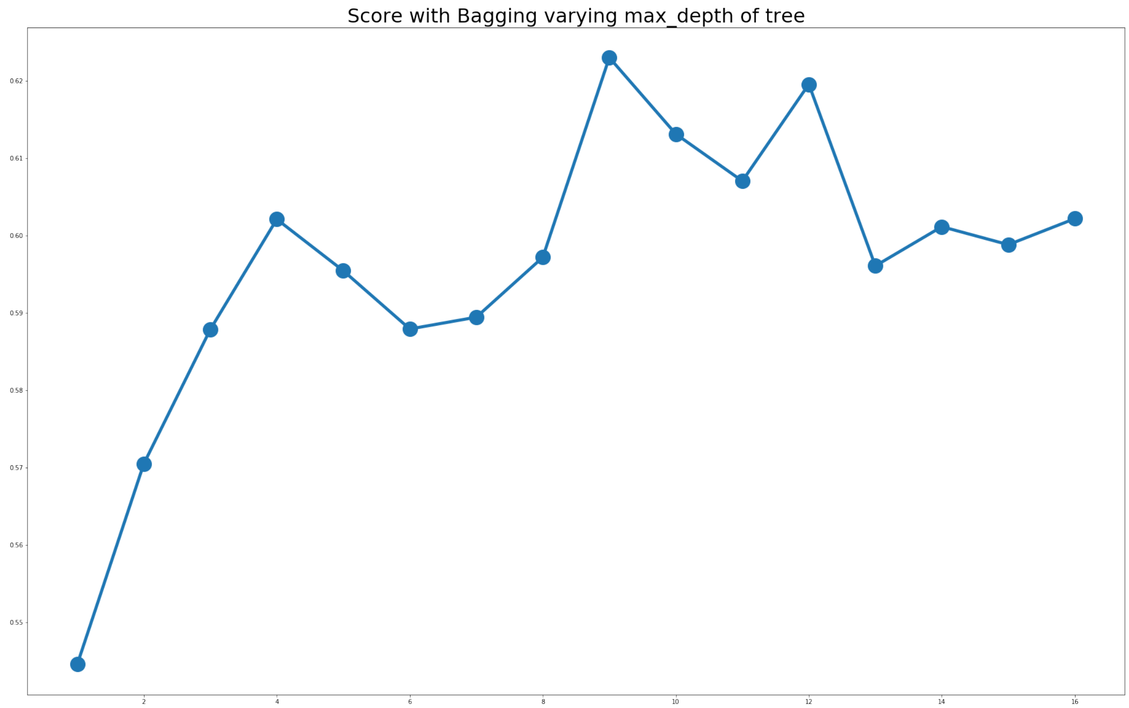
[22]:
```
scores_bagging = []
for par in parameter_values:
    estimator_bagging = BaggingClassifier(tree.
 ↪DecisionTreeClassifier(criterion="entropy"
                                            , max_depth = par
                                            )
                                          , max_samples=0.5
                                          , max_features=0.5
                                         )
    estimator_bagging.fit(X_train,y_train)
    scores = cross_val_score(estimator_bagging, X_train, y_train
                             , scoring='accuracy', cv = 5
                            )
    scores_bagging.append(np.mean(scores))
print(scores_bagging)
```

```
[0.5445619507841822, 0.570472844903677, 0.5878375361546816, 0.6020880744152197,
0.59542980042151, 0.5879047203227652, 0.5894282120448173, 0.5971717597546949,
0.6229901491890222, 0.613106868693959, 0.6070268680837339, 0.6195394112095709,
0.5960569201272726, 0.6011280851408065, 0.5988003934025972, 0.6021653693630428]
```

Plot the scores, as done in the previous cases

[23]:
```
plt.figure(figsize=(32,20))
plt.title("Score with Bagging varying max_depth of tree", fontsize = 32)
plt.plot(parameter_values
         , scores_bagging
```

```
        , '-o', linewidth=5
        , markersize=24
       );
```



2.0.1 Fit the tree after bagging and print summary

- store the parameter value giving the best score with `np.argmax(scores)`
- initialize an estimator as above, using the best parameter value computed above as maximum depth and `entropy` as criterion
- fit the estimator using the `train` part
- use the fitted estimator to predict using the test features
- compute the accuracy on the test and store it on a variable for the final summary
- print the accuracy on the test set and the best parameter value

[24]:
```
top_par_bagging = parameter_values[np.argmax(scores_bagging)]
estimator_bagging = BaggingClassifier(tree.
 ↪DecisionTreeClassifier(criterion="entropy"
                                              , max_depth = top_par_bagging
                                              )
                                            , max_samples=0.5
                                            , max_features=0.5
                                           )
```

```
estimator_bagging.fit(X_train,y_train); # the semicolon at the end prevents the␣
↪out[]
y_pred_bagging = estimator_bagging.predict(X_test)

accuracy_bagging = accuracy_score(y_pred_bagging, y_test)*100
print("The accuracy on test set tuned with bagging is {0:.1f}%".
↪format(accuracy_bagging))
print("Obtained with max_depth = {}".format(top_par_bagging))
```

```
The accuracy on test set tuned with bagging is 61.5%
Obtained with max_depth = 9
```

2.0.2 Final report

Print a summary of the four experiments

[25]:
```
print("                                        Accuracy   Hyperparameter")
print("Simple HoldOut and full tree        :   {:.1f}%      {}".
↪format(accuracy_ho, fitted_max_depth))
print("HoldOut and tuning on validation set:   {:.1f}%      {}".
↪format(accuracy_hov, top_par_hov))
print("CrossValidation and tuning          :   {:.1f}%      {}".
↪format(accuracy_cv, top_par_cv))
print("Ensemble Bagging and tuning         :   {:.1f}%      {}".
↪format(accuracy_bagging, top_par_bagging))
```

```
                                        Accuracy   Hyperparameter
Simple HoldOut and full tree        :   58.8%      16
HoldOut and tuning on validation set:   56.0%      13
CrossValidation and tuning          :   59.2%      15
Ensemble Bagging and tuning         :   61.5%      9
```

[26]:
```
import sklearn
print('The scikit-learn version is {}.'.format(sklearn.__version__))
```

```
The scikit-learn version is 0.21.3.
```

2.0.3 Suggested exercises

- try other datasets
- try to optimise the parameters ”min_impurity_decrease” ”min_samples_leaf” and ”min_samples_split”

ml-03-03-using-several-classifiers

February 12, 2020

1 Using several classifiers and tuning parameters - Parameters grid

From official `scikit-learn` documentation

Adapted by Claudio Sartori

Example of usage of the ***model selection*** features of `scikit-learn` and comparison of several classification methods. 1. import a sample dataset 1. split the dataset into two parts: train and test - the *train* part will be used for training and validation (i.e. for *development*) - the *test* part will be used for test (i.e. for *evaluation*) - the fraction of test data will be *ts* (a value of your choice between 0.2 and 0.5) 1. the function `GridSearchCV` iterates a cross validation experiment to train and test a model with different combinations of paramater values - for each parameter we set a list of values to test, the function will generate all the combinations - we choose a *score function* which will be used for the optimization - e.g. `accuracy_score`, `precision_score`, `cohen_kappa_score`, `f1_score`, see this page for reference - the output is a dictionary containing - the set of parameters which maximize the score - the test scores 1. prepare the parameters for the grid - it is a list of dictionaries 1. set the parameters by cross validation and the *score functions* to choose from 1. Loop on scores and, for each score, loop on the model labels (see details below)

[1]:
```
"""
http://scikit-learn.org/stable/auto_examples/model_selection/
 ↪plot_grid_search_digits.html
@author: scikit-learn.org and Claudio Sartori
"""
import warnings
warnings.filterwarnings('ignore') # uncomment this line to suppress warnings

from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.svm import SVC
from sklearn.linear_model import Perceptron
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
```

```
print(__doc__) # print information included in the triple quotes at the␣
 ↪beginning

# Loading a standard dataset
#dataset = datasets.load_digits()
#dataset = datasets.fetch_olivetti_faces()
#dataset = datasets.fetch_covtype()
dataset = datasets.load_iris()
#dataset = datasets.load_wine()
#dataset = datasets.load_breast_cancer()
```

```
http://scikit-
learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html
@author: scikit-learn.org and Claudio Sartori
```

1.0.1 Prepare the environment

The `dataset` module contains, among others, a few sample datasets.

See this page for reference

Prepare the data and the target in X and y. Set `ts`. Set the random state

[2]:
```
X = dataset.data
y = dataset.target
ts = 0.3
random_state = 42
```

Split the dataset into the train and test parts

[3]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=ts, random_state=random_state)
print("Training on %d examples" % len(X_train))
```

```
Training on 105 examples
```

The code below is intended to ease the remainder of the exercise

[4]:
```
model_lbls = [
#              'dt',
#              'nb',
#              'lp',
#              'svc',
             'knn',
            ]

# Set the parameters by cross-validation
```

```
tuned_param_dt = [{'max_depth': [range(1,20)]}]
tuned_param_nb = [{'var_smoothing': [10, 1, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6,␣
↪1e-07, 1e-8, 1e-9, 1e-10]}]
tuned_param_lp = [{'early_stopping': [True]}]
tuned_param_svc = [{'kernel': ['rbf'],
                    'gamma': [1e-3, 1e-4],
                    'C': [1, 10, 100, 1000],
                    },
                    {'kernel': ['linear'],
                     'C': [1, 10, 100, 1000],
                    },
                   ]
tuned_param_knn =[{'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}]

models = {
    'dt': {'name': 'Decision Tree       ',
           'estimator': DecisionTreeClassifier(),
           'param': tuned_param_dt,
          },
    'nb': {'name': 'Gaussian Naive Bayes',
           'estimator': GaussianNB(),
           'param': tuned_param_nb
          },
    'lp': {'name': 'Linear Perceptron   ',
           'estimator': Perceptron(),
           'param': tuned_param_lp,
          },
    'svc':{'name': 'Support Vector      ',
           'estimator': SVC(),
           'param': tuned_param_svc
          },
    'knn':{'name': 'K Nearest Neighbor ',
           'estimator': KNeighborsClassifier(),
           'param': tuned_param_knn

    }
}

scores = ['precision', 'recall']
```

1.0.2 The function below groups all the outputs

Write a function which has as parameter the fitted model and uses the components of the fitted model to inspect the results of the search with the parameters grid.

The components are: `model.best_params_` `model.cv_results_['mean_test_score']` `model.cv_results_['std` `model.cv_results_['params']`

The classification report is generated by the function imported above from sklearn.metrics, which takes as argument the true and the predicted test labels.

The +/- in the results is obtained doubling the `std_test_score`

The function will be used to print the results for each set of parameters

[5]:
```
def print_results(model):
    print("Best parameters set found on train set:")
    print()
    # if best is linear there is no gamma parameter
    print(model.best_params_)
    print()
    print("Grid scores on train set:")
    print()
    means = model.cv_results_['mean_test_score']
    stds = model.cv_results_['std_test_score']
    params = model.cv_results_['params']
    for mean, std, params_tuple in zip(means, stds, params):
        print("%0.3f (+/-%0.03f) for %r"
              % (mean, std * 2, params_tuple))
    print()
    print("Detailed classification report for the best parameter set:")
    print()
    print("The model is trained on the full train set.")
    print("The scores are computed on the full test set.")
    print()
    y_true, y_pred = y_test, model.predict(X_test)
    print(classification_report(y_true, y_pred))
    print()
```

1.0.3 Loop on scores and, for each score, loop on the model labels

- iterate varying the score function
    1. iterate varying the classification model among Decision Tree, Naive Bayes, Linear Perceptron, Support Vector
        - activate the *grid search*
            1. the resulting model will be the best one according to the current score function
        - print the best parameter set and the results for each set of parameters using the above defined function
        - print the classification report
        - store the `.best score_` in a dictionary for a final report
    2. print the final report for the current *score funtion*

[6]:
```
results_short = {}
```

[7]:
```
for score in scores:
    print('='*40)
    print("# Tuning hyper-parameters for %s" % score)
    print()

    #'%s_macro' % score ## is a string formatting expression
    # the parameter after % is substituted in the string placeholder %s
    for m in model_lbls:
        print('-'*40)
        print("Trying model {}".format(models[m]['name']))
        clf = GridSearchCV(models[m]['estimator'], models[m]['param'], cv=5,
                           scoring='%s_macro' % score,
                           iid = False,
                           return_train_score = False,
                           n_jobs = 2, # this allows using multi-cores
                           )
        clf.fit(X_train, y_train)
        print_results(clf)
        results_short[m] = clf.best_score_
    print("Summary of results for {}".format(score))
    print("Estimator")
    for m in results_short.keys():
        print("{}\t - score: {:4.2}%".format(models[m]['name'],␣
 ↪results_short[m]))
```

```
========================================
# Tuning hyper-parameters for precision

----------------------------------------
Trying model K Nearest Neighbor
Best parameters set found on train set:

{'n_neighbors': 8}

Grid scores on train set:

0.962 (+/-0.047) for {'n_neighbors': 1}
0.947 (+/-0.060) for {'n_neighbors': 2}
0.950 (+/-0.054) for {'n_neighbors': 3}
0.950 (+/-0.057) for {'n_neighbors': 4}
0.956 (+/-0.052) for {'n_neighbors': 5}
0.953 (+/-0.056) for {'n_neighbors': 6}
0.962 (+/-0.047) for {'n_neighbors': 7}
0.963 (+/-0.043) for {'n_neighbors': 8}
0.956 (+/-0.052) for {'n_neighbors': 9}
0.956 (+/-0.052) for {'n_neighbors': 10}
```

```
Detailed classification report for the best parameter set:

The model is trained on the full train set.
The scores are computed on the full test set.

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45


Summary of results for precision
Estimator
K Nearest Neighbor       - score: 0.96%
========================================
# Tuning hyper-parameters for recall

----------------------------------------
Trying model K Nearest Neighbor
Best parameters set found on train set:

{'n_neighbors': 8}

Grid scores on train set:

0.955 (+/-0.061) for {'n_neighbors': 1}
0.938 (+/-0.070) for {'n_neighbors': 2}
0.938 (+/-0.070) for {'n_neighbors': 3}
0.937 (+/-0.074) for {'n_neighbors': 4}
0.946 (+/-0.067) for {'n_neighbors': 5}
0.948 (+/-0.061) for {'n_neighbors': 6}
0.955 (+/-0.061) for {'n_neighbors': 7}
0.956 (+/-0.053) for {'n_neighbors': 8}
0.946 (+/-0.067) for {'n_neighbors': 9}
0.946 (+/-0.067) for {'n_neighbors': 10}

Detailed classification report for the best parameter set:

The model is trained on the full train set.
The scores are computed on the full test set.

              precision    recall  f1-score   support
```

```
           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45


Summary of results for recall
Estimator
K Nearest Neighbor       - score: 0.96%
```

ml-03-04-classif-w-prepr

February 12, 2020

1 Preprocessing: transform categorical data

In `scikit-learn` the classifiers require numeric data. The library makes available a set of preprocessing fuctions which help the transformation. This exercise proposes two types of transformations:

- `OneHotEncoder` for purely categorical columns: if the column has **V** distinct values it is substituted by **V** binary columns where in each row only the bit corrosponding to the original value is true
- `OrdinalEncoder` for ordinal columns: the original **V** values are mapped into the **0..V-1** range

The additional function `ColumnTransformer` allows to apply the different transformations to the appropriate columns with a single statement.

1.0.1 To do:

- import the appropriate names
- set the random state
- import the data set with the appropriate column names
- inspect the content and the data types
- read carefully the `.names` file of the data set, to understand which are the ordinal and categorical data
- data cleaning
  - the **ordinal transformer** generates a mapping from strings to numbers according to the lexicographic sorting of the strings; in this particular case, the strings indicate numeric subranges, and ranges with one digit constitute exceptions '5-9' happens to be after '20-25'
  - it is necessary to transform '5-9' into '05-09', and the same for other similar cases
  - a way to do this is to prepare dictionaries for the translation and use the `.map` function
- prepare the lists of the ordinal, categorical and numeric columns
- prepare the preprocessor
- split the cleaned data into the X and y part
- fit_transform the preprocessor and generate the transformed data set
- split the transformed data set into train and test
- use the same method used for the exercise of 19/11 to test several classifiers

[1]:
```
"""
http://scikit-learn.org/stable/auto_examples/model_selection/
→plot_grid_search_digits.html
```

```
@author: scikit-learn.org and Claudio Sartori
"""
import warnings
warnings.filterwarnings('ignore') # uncomment this line to suppress warnings

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV

from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder
from sklearn.compose import ColumnTransformer

from sklearn.svm import SVC
from sklearn.linear_model import Perceptron
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier

print(__doc__) # print information included in the triple quotes at the␣
↪beginning

random_state = 42
```

http://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html
@author: scikit-learn.org and Claudio Sartori

[2]:
```
# url = 'diagnosis.data'
# names = ['Temp', 'Nau', 'Lum', 'Uri', 'Mic', 'Bur', 'd1', 'd2']
# sep = "\t"
url = 'breast-cancer.data'
names = ['Class','age','menopause','tumor-size','inv-nodes',
         'node-caps','deg-malig','breast','breast-quad','irradiat']
sep = ","

df = pd.read_csv(url, names = names, sep=sep)
df.head()
```

[2]:

```
                  Class    age menopause tumor-size inv-nodes node-caps  \
0  no-recurrence-events  30-39   premeno      30-34       0-2        no
1  no-recurrence-events  40-49   premeno      20-24       0-2        no
2  no-recurrence-events  40-49   premeno      20-24       0-2        no
3  no-recurrence-events  60-69      ge40      15-19       0-2        no
4  no-recurrence-events  40-49   premeno        0-4       0-2        no

   deg-malig breast breast-quad irradiat
0          3   left    left_low       no
1          2  right    right_up       no
2          2   left    left_low       no
3          2  right     left_up       no
4          2  right   right_low       no
```

Show the types of the columns

[3]:

```
print(df.dtypes)
```

```
Class          object
age            object
menopause      object
tumor-size     object
inv-nodes      object
node-caps      object
deg-malig       int64
breast         object
breast-quad    object
irradiat       object
dtype: object
```

Clean the column `tumor-size`

[4]:

```
tumor_size_dict = dict(zip(list(df['tumor-size'].
 ↪unique()),list(df['tumor-size'].unique())))
tumor_size_dict
```

[4]:

```
{'30-34': '30-34',
 '20-24': '20-24',
 '15-19': '15-19',
 '0-4': '0-4',
 '25-29': '25-29',
 '50-54': '50-54',
 '10-14': '10-14',
 '40-44': '40-44',
 '35-39': '35-39',
 '5-9': '5-9',
 '45-49': '45-49'}
```

[5]:
```
tumor_size_dict['0-4'] = '00-04'
tumor_size_dict['5-9'] = '05-09'
```

[6]:
```
df['tumor-size'] = df['tumor-size'].map(tumor_size_dict)
```

Clean the column `inv-nodes`

[7]:
```
inv_nodes_dict = dict(zip(list(df['inv-nodes'].unique()),list(df['inv-nodes'].
 ↪unique())))
```

[8]:
```
inv_nodes_dict['0-2']  = '00-02'
inv_nodes_dict['3-5']  = '03-05'
inv_nodes_dict['6-8']  = '06-08'
inv_nodes_dict['9-11'] = '09-11'
```

[9]:
```
df['inv-nodes'] = df['inv-nodes'].map(inv_nodes_dict)
```

Inspect the data

[10]:
```
df.head()
```

[10]:
```
                  Class    age menopause tumor-size inv-nodes node-caps  \
0  no-recurrence-events  30-39   premeno      30-34     00-02        no
1  no-recurrence-events  40-49   premeno      20-24     00-02        no
2  no-recurrence-events  40-49   premeno      20-24     00-02        no
3  no-recurrence-events  60-69      ge40      15-19     00-02        no
4  no-recurrence-events  40-49   premeno      00-04     00-02        no

   deg-malig breast breast-quad irradiat
0          3   left    left_low       no
1          2  right    right_up       no
2          2   left    left_low       no
3          2  right     left_up       no
4          2  right   right_low       no
```

Prepare the lists of numeric features, ordinal features, categorical features

[11]:
```
categorical_features = df.dtypes.loc[df.dtypes == 'object'].index.values
print("The non-numeric features are:")
print(categorical_features)
```

```
The non-numeric features are:
['Class' 'age' 'menopause' 'tumor-size' 'inv-nodes' 'node-caps' 'breast'
 'breast-quad' 'irradiat']
```

[12]:
```
numeric_features = list(set(df.dtypes.index.values)-set(categorical_features))
print("The numeric features are:")
print(numeric_features)
```

```
The numeric features are:
['deg-malig']
```

[13]:
```
ordinal_features =['age', 'tumor-size','inv-nodes']
print("The ordinal features are:")
print(ordinal_features)
```

```
The ordinal features are:
['age', 'tumor-size', 'inv-nodes']
```

[14]:
```
categorical_features = list(set(categorical_features) - set(ordinal_features) -␣
↪set(['Class']))
print("The categorical features are:")
print(categorical_features)
```

```
The categorical features are:
['menopause', 'irradiat', 'breast', 'node-caps', 'breast-quad']
```

Prepare the transformer

[15]:
```
# transf_dtype = np.float64
transf_dtype = np.int32

categorical_transformer = OneHotEncoder(handle_unknown='ignore', sparse =␣
↪False, dtype = transf_dtype)
ordinal_transformer = OrdinalEncoder(dtype = transf_dtype)
preprocessor = ColumnTransformer(
    transformers = [('cat', categorical_transformer, categorical_features),
                    ('ord', ordinal_transformer, ordinal_features)
                   ],
                    remainder = 'passthrough'
    )
```

Split X and y and check the shapes

[16]:
```
X = df.drop(['Class'], axis = 1)
y = df['Class']
```

[17]:
```
labels = y.unique()
print("The labels are:")
print(labels)
```

```
The labels are:
['no-recurrence-events' 'recurrence-events']
```

[18]:
```
X.shape
```

[18]: (286, 9)

Fit the preprocessor with X and check the parameters printing the `.named_transformers_` attribute

[19]:
```
preprocessor.fit(X)
```

[19]:
```
ColumnTransformer(n_jobs=None, remainder='passthrough', sparse_threshold=0.3,
                  transformer_weights=None,
                  transformers=[('cat',
                                 OneHotEncoder(categorical_features=None,
                                               categories=None, drop=None,
                                               dtype=<class 'numpy.int32'>,
                                               handle_unknown='ignore',
                                               n_values=None, sparse=False),
                                 ['menopause', 'irradiat', 'breast',
                                  'node-caps', 'breast-quad']),
                                ('ord',
                                 OrdinalEncoder(categories='auto',
                                                dtype=<class 'numpy.int32'>),
                                 ['age', 'tumor-size', 'inv-nodes'])],
                  verbose=False)
```

[20]:
```
print(preprocessor.named_transformers_)
```

```
{'cat': OneHotEncoder(categorical_features=None, categories=None, drop=None,
              dtype=<class 'numpy.int32'>, handle_unknown='ignore',
              n_values=None, sparse=False), 'ord':
OrdinalEncoder(categories='auto', dtype=<class 'numpy.int32'>), 'remainder':
'passthrough'}
```

Fit-transform X and store the result in X_p, check the shape

[21]:
```
X_p = preprocessor.fit_transform(X)
```

[22]:
```
X_p.shape
```

[22]:
```
(286, 20)
```

For ease of inspection transform `X_p` into a data frame `df_p` and inspect it

[23]:
```
df_p = pd.DataFrame(X_p)
```

[24]:
```
df_p.describe()
```

[24]:

	0	1	2	3	4	5 \
count	286.000000	286.000000	286.000000	286.000000	286.000000	286.000000
mean	0.451049	0.024476	0.524476	0.762238	0.237762	0.531469
std	0.498470	0.154791	0.500276	0.426459	0.426459	0.499883
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	1.000000	0.000000	0.000000

```
50%        0.000000    0.000000    1.000000    1.000000    0.000000    1.000000
75%        1.000000    0.000000    1.000000    1.000000    0.000000    1.000000
max        1.000000    1.000000    1.000000    1.000000    1.000000    1.000000

                6           7           8           9          10          11  \
count  286.000000  286.000000  286.000000  286.000000  286.000000  286.000000
mean     0.468531    0.027972    0.776224    0.195804    0.003497    0.073427
std      0.499883    0.165182    0.417504    0.397514    0.059131    0.261293
min      0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
25%      0.000000    0.000000    1.000000    0.000000    0.000000    0.000000
50%      0.000000    0.000000    1.000000    0.000000    0.000000    0.000000
75%      1.000000    0.000000    1.000000    0.000000    0.000000    0.000000
max      1.000000    1.000000    1.000000    1.000000    1.000000    1.000000

               12          13          14          15          16          17  \
count  286.000000  286.000000  286.000000  286.000000  286.000000  286.000000
mean     0.384615    0.339161    0.083916    0.115385    2.664336    4.881119
std      0.487357    0.474254    0.277748    0.320046    1.011818    2.105930
min      0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
25%      0.000000    0.000000    0.000000    0.000000    2.000000    4.000000
50%      0.000000    0.000000    0.000000    0.000000    3.000000    5.000000
75%      1.000000    1.000000    0.000000    0.000000    3.000000    6.000000
max      1.000000    1.000000    1.000000    1.000000    5.000000   10.000000

               18          19
count  286.000000  286.000000
mean     0.517483    2.048951
std      1.110417    0.738217
min      0.000000    1.000000
25%      0.000000    2.000000
50%      0.000000    2.000000
75%      1.000000    3.000000
max      6.000000    3.000000
```

[25]:

```
df_p.head()
```

[25]:

```
   0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19
0  0  0  1  1  0  1  0  0  1  0   0   0   1   0   0   0   1   6   0   3
1  0  0  1  1  0  0  1  0  1  0   0   0   0   0   0   1   2   4   0   2
2  0  0  1  1  0  1  0  0  1  0   0   0   1   0   0   0   2   4   0   2
3  1  0  0  1  0  0  1  0  1  0   0   0   0   1   0   0   4   3   0   2
4  0  0  1  1  0  0  1  0  1  0   0   0   0   0   1   0   2   0   0   2
```

The columns in the transformed dataset are generated according to the order you see printing the preprocessor after fitting, therefore the last four columns correspond to `'age'`, `'tumor-size'`, `'inv-nodes'`, `'deg-malig'`.

In order to inspect if the translation and check if the mapping is as expected, compare the sorted values of df['tumor-size'] and df_p[17], e.g. comparing the index sequences

[26]:
```
orig_col = 'tumor-size'
transf_col = 17
a=pd.DataFrame(zip(df[orig_col],df_p[transf_col]))
print('The number of index discordances between \'{}\' and \'{}\' is {}'.\
      format(orig_col, transf_col, sum(a.sort_values(by = 0).index.values!=a.
 ↪sort_values(by = 1).index.values)))
```

```
The number of index discordances between 'tumor-size' and '17' is 0
```

Train/test split

[27]:
```
X_train, X_test, y_train, y_test = train_test_split(X_p,y, random_state =␣
 ↪random_state)
```

Classification and test

[28]:
```
model_lbls = [
             'dt',
             'nb',
             'lp',
             'svc',
              'knn',
             'rfc',
             'ada',
            ]

# Set the parameters by cross-validation
tuned_param_dt = [{'max_depth': list(range(1,20))}]
tuned_param_nb = [{'var_smoothing': [10**i for i in range(1,-11, -1)]}]
tuned_param_lp = [{'early_stopping': [True]}]
tuned_param_svc = [{'kernel': ['rbf'],
                    'gamma': [1e-3, 1e-4],
#                     'C': [1, 10, 100, 1000],
                    'C': [10**i for i in range(0,4)],
                    },
                    {'kernel': ['linear'],
                    'C': [10**i for i in range(0,4)],
                    },
                   ]
tuned_param_knn =[{'n_neighbors': list(range(1,11)),
                   'metric': ['euclidean', 'manhattan', 'chebyshev']}
                 ]
tuned_param_rfc =[{'max_depth': list(range(1,11))}]
tuned_param_ada = [{'learning_rate': [1., 0.1, 0.01, 0.001, 0.0001]}]

models = {
    'dt': {'name': 'Decision Tree       ',
```

```
            'estimator': DecisionTreeClassifier(),
            'param': tuned_param_dt,
           },
    'nb': {'name': 'Gaussian Naive Bayes',
            'estimator': GaussianNB(),
            'param': tuned_param_nb
           },
    'lp': {'name': 'Linear Perceptron   ',
            'estimator': Perceptron(),
            'param': tuned_param_lp,
           },
    'svc':{'name': 'Support Vector      ',
            'estimator': SVC(),
            'param': tuned_param_svc
           },
    'knn':{'name': 'K Nearest Neighbor  ',
            'estimator': KNeighborsClassifier(),
            'param': tuned_param_knn
           },
    'rfc':{'name': 'Random Forest       ',
            'estimator': RandomForestClassifier(),
            'param': tuned_param_rfc
           },
    'ada':{'name': 'Adaboost            ',
            'estimator': AdaBoostClassifier(),
            'param': tuned_param_ada
           },
}

scores = [
#    'precision_macro',
    'recall_macro',
#    'accuracy',
#    'f1_macro'
]
```

[29]:
```
# def plot_confusion_matrix(cm):
#     print(cm)
#     fig = plt.figure(figsize=(10,10))
#     ax = fig.add_subplot(111)
#     cax = ax.matshow(cm)
#     plt.title('Confusion matrix of the classifier')
#     fig.colorbar(cax)
#     ax.set_xticklabels([''] + labels)
#     ax.set_yticklabels([''] + labels)
#     plt.xlabel('Predicted')
#     plt.ylabel('True')
```

```python
#     plt.show()

def print_results(model):
    print("Best parameters set found on train set:")
    print()
    # if best is linear there is no gamma parameter
    print(model.best_params_)
    print()
    print("Grid scores on train set:")
    print()
    means = model.cv_results_['mean_test_score']
    stds = model.cv_results_['std_test_score']
    params = model.cv_results_['params']
    for mean, std, params_tuple in zip(means, stds, params):
        print("%0.3f (+/-%0.03f) for %r"
              % (mean, std * 2, params_tuple))
    print()
    print("Detailed classification report for the best parameter set:")
    print()
    print("The model is trained on the full train set.")
    print("The scores are computed on the full test set.")
    print()
    y_true, y_pred = y_test, model.predict(X_test)
    print(classification_report(y_true, y_pred))
    cm = confusion_matrix(y_true,y_pred, labels = labels)
    print(cm)
#     plot_confusion_matrix(cm)
    print()
```

[30]:
```python
results_short = {}

for score in scores:
    print('='*40)
    print("# Tuning hyper-parameters for %s" % score)
    print()

    #'%s_macro' % score ## is a string formatting expression
    # the parameter after % is substituted in the string placeholder %s
    for m in model_lbls:
        print('-'*40)
        print("Trying model {}".format(models[m]['name']))
        clf = GridSearchCV(models[m]['estimator'], models[m]['param'], cv=5,
                           scoring=score,
                           iid = False,
                           return_train_score = False,
                           n_jobs = 2, # this allows using multi-cores
                           )
```

```
        clf.fit(X_train, y_train)
        print_results(clf)
        results_short[m] = clf.best_score_
    print("Summary of results for {}".format(score))
    print("Estimator")
    for m in results_short.keys():
        print("{}\t - score: {:4.2}%".format(models[m]['name'],␣
↪results_short[m]))
```

```
========================================
# Tuning hyper-parameters for recall_macro

----------------------------------------
Trying model Decision Tree
Best parameters set found on train set:

{'max_depth': 14}

Grid scores on train set:

0.567 (+/-0.086) for {'max_depth': 1}
0.610 (+/-0.115) for {'max_depth': 2}
0.583 (+/-0.127) for {'max_depth': 3}
0.551 (+/-0.082) for {'max_depth': 4}
0.574 (+/-0.148) for {'max_depth': 5}
0.574 (+/-0.138) for {'max_depth': 6}
0.597 (+/-0.148) for {'max_depth': 7}
0.591 (+/-0.226) for {'max_depth': 8}
0.567 (+/-0.223) for {'max_depth': 9}
0.576 (+/-0.285) for {'max_depth': 10}
0.577 (+/-0.168) for {'max_depth': 11}
0.552 (+/-0.166) for {'max_depth': 12}
0.564 (+/-0.163) for {'max_depth': 13}
0.620 (+/-0.187) for {'max_depth': 14}
0.565 (+/-0.142) for {'max_depth': 15}
0.573 (+/-0.089) for {'max_depth': 16}
0.608 (+/-0.121) for {'max_depth': 17}
0.571 (+/-0.154) for {'max_depth': 18}
0.576 (+/-0.177) for {'max_depth': 19}

Detailed classification report for the best parameter set:

The model is trained on the full train set.
The scores are computed on the full test set.

                      precision    recall  f1-score   support
```

```
no-recurrence-events       0.72      0.84      0.77        49
   recurrence-events       0.47      0.30      0.37        23

            accuracy                           0.67        72
           macro avg       0.59      0.57      0.57        72
        weighted avg       0.64      0.67      0.64        72

[[41  8]
 [16  7]]

----------------------------------------
Trying model Gaussian Naive Bayes
Best parameters set found on train set:

{'var_smoothing': 0.01}

Grid scores on train set:

0.500 (+/-0.000) for {'var_smoothing': 10}
0.506 (+/-0.049) for {'var_smoothing': 1}
0.593 (+/-0.115) for {'var_smoothing': 0.1}
0.629 (+/-0.134) for {'var_smoothing': 0.01}
0.627 (+/-0.125) for {'var_smoothing': 0.001}
0.624 (+/-0.121) for {'var_smoothing': 0.0001}
0.611 (+/-0.076) for {'var_smoothing': 1e-05}
0.601 (+/-0.092) for {'var_smoothing': 1e-06}
0.591 (+/-0.094) for {'var_smoothing': 1e-07}
0.577 (+/-0.124) for {'var_smoothing': 1e-08}
0.556 (+/-0.142) for {'var_smoothing': 1e-09}
0.551 (+/-0.135) for {'var_smoothing': 1e-10}

Detailed classification report for the best parameter set:

The model is trained on the full train set.
The scores are computed on the full test set.

                      precision    recall  f1-score   support

no-recurrence-events       0.73      0.88      0.80        49
   recurrence-events       0.54      0.30      0.39        23

            accuracy                           0.69        72
           macro avg       0.63      0.59      0.59        72
        weighted avg       0.67      0.69      0.67        72

[[43  6]
 [16  7]]
```

```
----------------------------------------
Trying model Linear Perceptron
Best parameters set found on train set:

{'early_stopping': True}

Grid scores on train set:

0.564 (+/-0.111) for {'early_stopping': True}

Detailed classification report for the best parameter set:

The model is trained on the full train set.
The scores are computed on the full test set.

                      precision    recall  f1-score   support

no-recurrence-events       1.00      0.14      0.25        49
   recurrence-events       0.35      1.00      0.52        23

            accuracy                           0.42        72
           macro avg       0.68      0.57      0.39        72
        weighted avg       0.79      0.42      0.34        72

[[ 7 42]
 [ 0 23]]

----------------------------------------
Trying model Support Vector
Best parameters set found on train set:

{'C': 10, 'kernel': 'linear'}

Grid scores on train set:

0.500 (+/-0.000) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}
0.500 (+/-0.000) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.495 (+/-0.048) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
0.500 (+/-0.000) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.549 (+/-0.064) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}
0.495 (+/-0.048) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
0.574 (+/-0.122) for {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
0.554 (+/-0.074) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.582 (+/-0.091) for {'C': 1, 'kernel': 'linear'}
0.599 (+/-0.159) for {'C': 10, 'kernel': 'linear'}
0.599 (+/-0.159) for {'C': 100, 'kernel': 'linear'}
0.599 (+/-0.159) for {'C': 1000, 'kernel': 'linear'}
```

```
Detailed classification report for the best parameter set:

The model is trained on the full train set.
The scores are computed on the full test set.

                      precision    recall  f1-score   support

no-recurrence-events       0.73      0.92      0.81        49
   recurrence-events       0.60      0.26      0.36        23

            accuracy                           0.71        72
           macro avg       0.66      0.59      0.59        72
        weighted avg       0.69      0.71      0.67        72

[[45  4]
 [17  6]]

----------------------------------------
Trying model K Nearest Neighbor
Best parameters set found on train set:

{'metric': 'manhattan', 'n_neighbors': 7}

Grid scores on train set:

0.567 (+/-0.088) for {'metric': 'euclidean', 'n_neighbors': 1}
0.524 (+/-0.037) for {'metric': 'euclidean', 'n_neighbors': 2}
0.554 (+/-0.190) for {'metric': 'euclidean', 'n_neighbors': 3}
0.542 (+/-0.104) for {'metric': 'euclidean', 'n_neighbors': 4}
0.548 (+/-0.115) for {'metric': 'euclidean', 'n_neighbors': 5}
0.502 (+/-0.075) for {'metric': 'euclidean', 'n_neighbors': 6}
0.555 (+/-0.096) for {'metric': 'euclidean', 'n_neighbors': 7}
0.523 (+/-0.080) for {'metric': 'euclidean', 'n_neighbors': 8}
0.521 (+/-0.091) for {'metric': 'euclidean', 'n_neighbors': 9}
0.524 (+/-0.079) for {'metric': 'euclidean', 'n_neighbors': 10}
0.578 (+/-0.077) for {'metric': 'manhattan', 'n_neighbors': 1}
0.554 (+/-0.063) for {'metric': 'manhattan', 'n_neighbors': 2}
0.554 (+/-0.170) for {'metric': 'manhattan', 'n_neighbors': 3}
0.570 (+/-0.086) for {'metric': 'manhattan', 'n_neighbors': 4}
0.562 (+/-0.052) for {'metric': 'manhattan', 'n_neighbors': 5}
0.553 (+/-0.097) for {'metric': 'manhattan', 'n_neighbors': 6}
0.584 (+/-0.124) for {'metric': 'manhattan', 'n_neighbors': 7}
0.566 (+/-0.158) for {'metric': 'manhattan', 'n_neighbors': 8}
0.560 (+/-0.161) for {'metric': 'manhattan', 'n_neighbors': 9}
0.561 (+/-0.095) for {'metric': 'manhattan', 'n_neighbors': 10}
0.490 (+/-0.152) for {'metric': 'chebyshev', 'n_neighbors': 1}
0.521 (+/-0.128) for {'metric': 'chebyshev', 'n_neighbors': 2}
0.575 (+/-0.146) for {'metric': 'chebyshev', 'n_neighbors': 3}
```

```
0.539 (+/-0.090) for {'metric': 'chebyshev', 'n_neighbors': 4}
0.576 (+/-0.095) for {'metric': 'chebyshev', 'n_neighbors': 5}
0.518 (+/-0.087) for {'metric': 'chebyshev', 'n_neighbors': 6}
0.531 (+/-0.100) for {'metric': 'chebyshev', 'n_neighbors': 7}
0.518 (+/-0.068) for {'metric': 'chebyshev', 'n_neighbors': 8}
0.520 (+/-0.086) for {'metric': 'chebyshev', 'n_neighbors': 9}
0.539 (+/-0.070) for {'metric': 'chebyshev', 'n_neighbors': 10}

Detailed classification report for the best parameter set:

The model is trained on the full train set.
The scores are computed on the full test set.

                      precision    recall  f1-score   support

no-recurrence-events       0.69      0.92      0.79        49
   recurrence-events       0.43      0.13      0.20        23

            accuracy                           0.67        72
           macro avg       0.56      0.52      0.49        72
        weighted avg       0.61      0.67      0.60        72

[[45  4]
 [20  3]]

----------------------------------------
Trying model Random Forest
Best parameters set found on train set:

{'max_depth': 8}

Grid scores on train set:

0.533 (+/-0.082) for {'max_depth': 1}
0.604 (+/-0.076) for {'max_depth': 2}
0.587 (+/-0.130) for {'max_depth': 3}
0.595 (+/-0.111) for {'max_depth': 4}
0.599 (+/-0.194) for {'max_depth': 5}
0.590 (+/-0.115) for {'max_depth': 6}
0.580 (+/-0.117) for {'max_depth': 7}
0.614 (+/-0.110) for {'max_depth': 8}
0.560 (+/-0.087) for {'max_depth': 9}
0.600 (+/-0.112) for {'max_depth': 10}

Detailed classification report for the best parameter set:

The model is trained on the full train set.
The scores are computed on the full test set.
```

```
                      precision    recall  f1-score   support

no-recurrence-events       0.73      0.96      0.83        49
   recurrence-events       0.75      0.26      0.39        23

            accuracy                           0.74        72
           macro avg       0.74      0.61      0.61        72
        weighted avg       0.74      0.74      0.69        72

[[47  2]
 [17  6]]

----------------------------------------
Trying model Adaboost
Best parameters set found on train set:

{'learning_rate': 0.01}

Grid scores on train set:

0.586 (+/-0.146) for {'learning_rate': 1.0}
0.620 (+/-0.102) for {'learning_rate': 0.1}
0.643 (+/-0.161) for {'learning_rate': 0.01}
0.567 (+/-0.086) for {'learning_rate': 0.001}
0.567 (+/-0.086) for {'learning_rate': 0.0001}

Detailed classification report for the best parameter set:

The model is trained on the full train set.
The scores are computed on the full test set.

                      precision    recall  f1-score   support

no-recurrence-events       0.72      0.98      0.83        49
   recurrence-events       0.80      0.17      0.29        23

            accuracy                           0.72        72
           macro avg       0.76      0.58      0.56        72
        weighted avg       0.74      0.72      0.65        72

[[48  1]
 [19  4]]

Summary of results for recall_macro
Estimator
Decision Tree            - score: 0.62%
Gaussian Naive Bayes     - score: 0.63%
```

```
Linear Perceptron        - score: 0.56%
Support Vector           - score:  0.6%
K Nearest Neighbor       - score: 0.58%
Random Forest            - score: 0.61%
Adaboost                 - score: 0.64%
```

ml-04-ex1-KMeans-elbow

February 12, 2020

Claudio Sartori

Elaboration from the example given in Sebastian Raschka, 2015

https://github.com/rasbt/python-machine-learning-book

1 Machine Learning - Lab

1.1 Working with Unlabeled Data – Clustering Analysis

1.1.1 Find the best number of clusters with k_means

1.1.2 Overview

- Section 2
- Section 2.1
- Section 2.2

[1]:
```
from IPython.display import Image
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, silhouette_samples

%matplotlib inline

rnd_state = 42 # This variable will be used in all the procedure calls allowing␣
↪a random_state parameter
                # in this way the running can be perfectly reproduced
                # just change this value for a different experiment
```

2 Grouping objects by similarity using k-means

In this example we will use an *artificial* data set

1. load the data file from `'ex1_4dim_data.csv'`
2. check the shape and plot the content

3. observe the plot and decide which are the most interesting columns, to use in the plots of the clusters

- make a 2d plot of the two most promising columns

4. Use the elbow method to find the optimal number of clusters: test `KMeans` with varying number of clusters, from 2 to 10, fitting the data and computing the inertia and the silhouette score

5. Choose the optimal number of clusters looking at the plots, then cluster the data, plot the clusters and plot the scores of the individual samples
6. For comparison, repeat 5 with two clusters

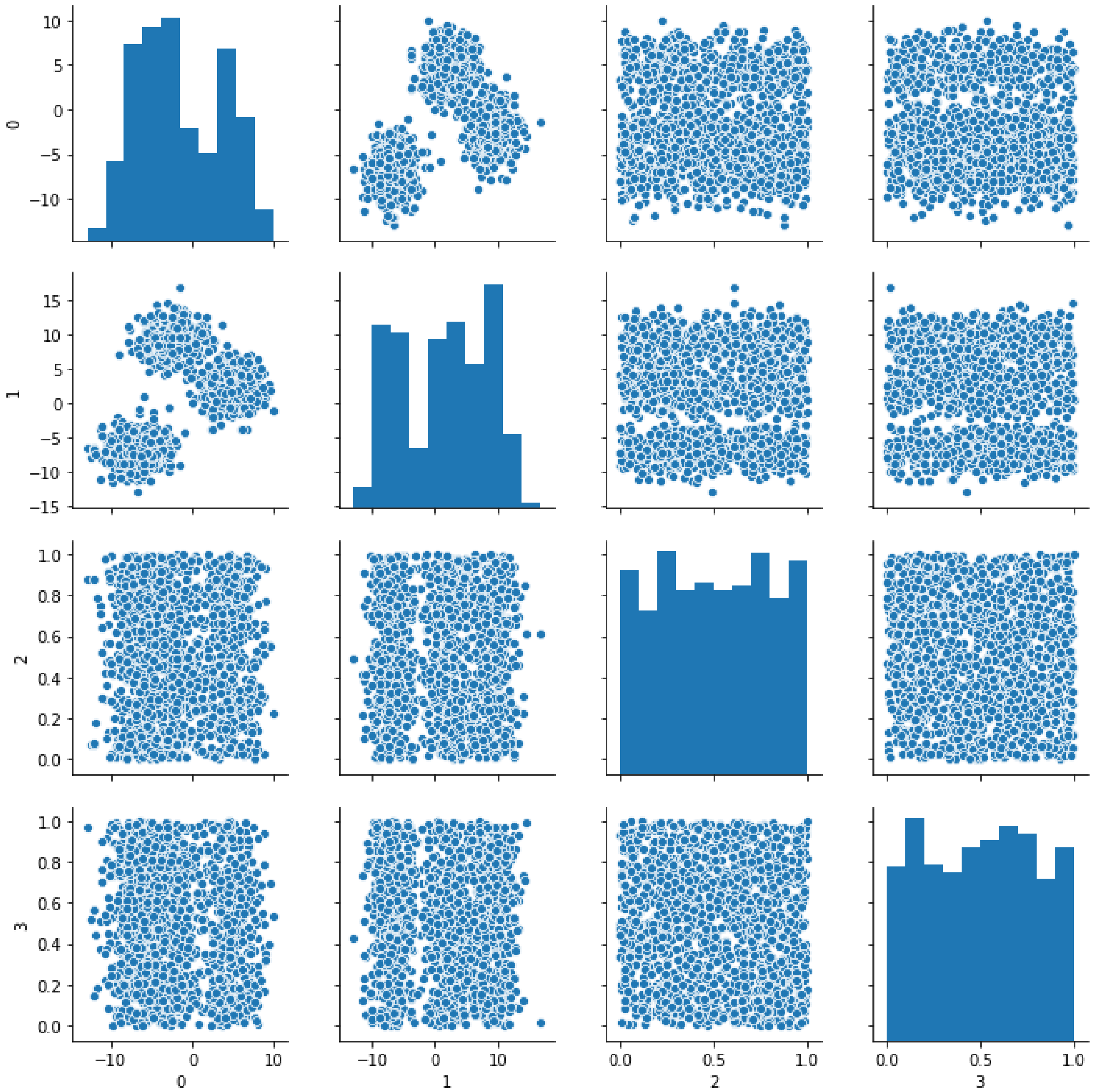
[2]:
```
data_file = 'ex1_4dim_data.csv'
delimiter = ','
X = np.loadtxt(data_file, delimiter = delimiter)
```

[3]:
```
X.shape
```

[3]:
```
(1500, 4)
```

[4]:
```
sns.pairplot(pd.DataFrame(X))
```

[4]:
```
<seaborn.axisgrid.PairGrid at 0x1a242bbfd0>
```

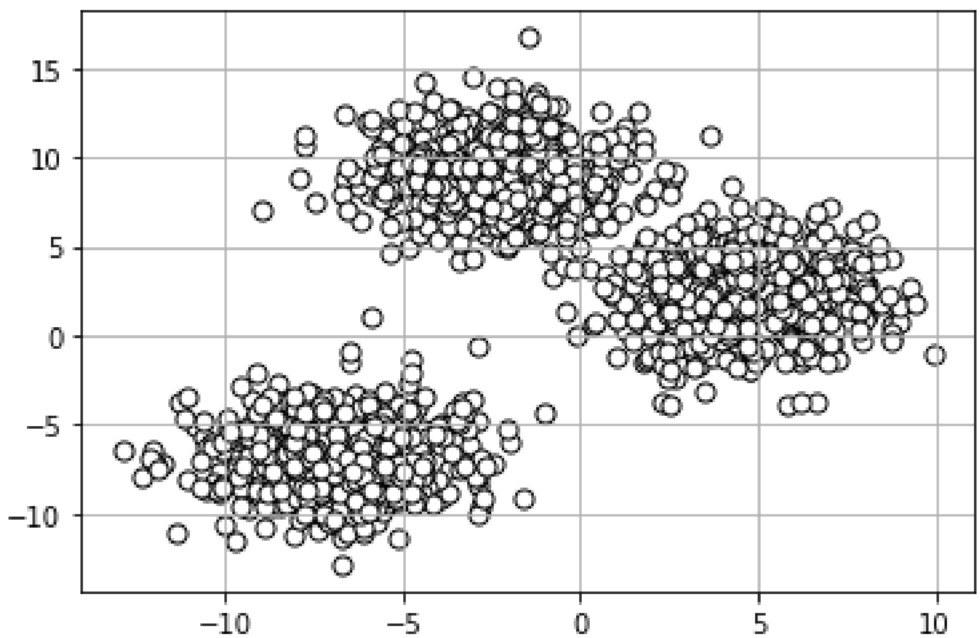


2.0.1 3. Observe the pairplots

In this simple example you can easily see that the two most interesting columns are 0 and 1.

[5]:
```
focus = [0,1]
plt.scatter(X[:,focus[0]], X[:,focus[1]]
            , c='white'           # color filling the data markers
            , edgecolors='black' # edge color for data markers
            , marker='o'          # data marker shape, e.g. triangles (v<>^),␣
 ↪square (s), star (*), ...
            , s=50)               # data marker size
plt.grid()  # plots a grid on the data
```

```
plt.show()
```



[6]:
```
from plot_clusters import plot_clusters
```

[7]:
```
help(plot_clusters)
```

```
Help on function plot_clusters in module plot_clusters:

plot_clusters(X, y, dim, points, labels_prefix='cluster',
points_name='centroids', colors=<matplotlib.colors.ListedColormap object at
0x11b0e5b50>, points_color=(0.09019607843137255, 0.7450980392156863,
0.8117647058823529, 1.0))
    Plot a two dimensional projection of an array of labelled points
    X:      array with at least two columns
    y:      vector of labels, length as number of rows in X
    dim:    the two columns to project, inside range of X columns, e.g. (0,1)
    points: additional points to plot as 'stars'
    labels_prefix: prefix to the labels for the legend ['cluster']
    points_name:   legend name for the additional points ['centroids']
    colors: a color map
    points_color: the color for the points
```

2.1 Using the elbow method to find the optimal number of clusters

We will try **k_means** with a number of clusters varying from 2 to 10

- prepare two emptys lists for inertia and silhouette scores
- For each value of the number of clusters:

- initialize an estimator for `KMeans` and `fit_predict`
- we will store the distortion (from the fitted model) in the variable `distortions`
- using the function `silhouette_score` from `sklearn.metrics` with arguments the data and the fitted labels, we will fill the variable `silhouette_scores`

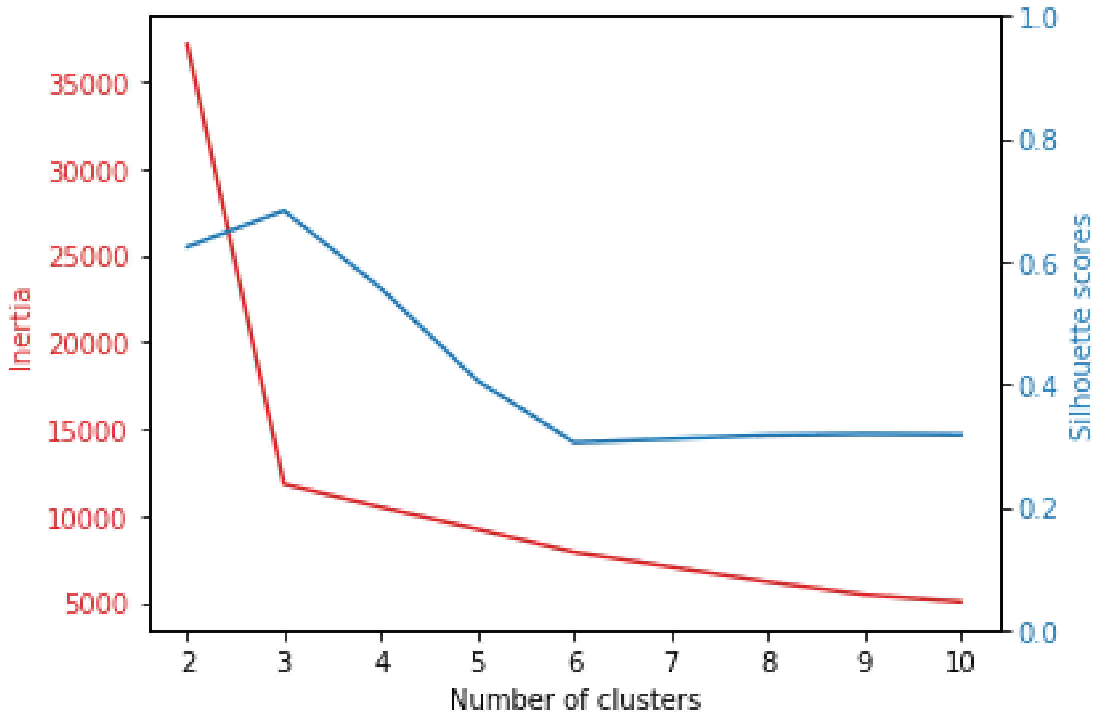
Then we will plot the two lists in the y axis, with the range of k in the x axis. The plot with two different scales in the `y` axis can be done according to the example shown in the notebook `two_scales.ipynb`.

[8]:
```
k_range = range(2,11)
```

[9]:
```
distortions = []
silhouette_scores = []
for i in k_range:
    km = KMeans(n_clusters=i,
                init='k-means++',
                n_init=10,
                max_iter=300,
                random_state=rnd_state)
    y_km = km.fit_predict(X)
    distortions.append(km.inertia_)
    silhouette_scores.append(silhouette_score(X,y_km))
```

[10]:
```
fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('Number of clusters')
ax1.set_ylabel('Inertia', color=color)
ax1.plot(k_range, distortions, color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('Silhouette scores', color=color)  # we already handled the␣
 ↪x-label with ax1
ax2.plot(k_range, silhouette_scores, color=color)
ax2.tick_params(axis='y', labelcolor=color)
ax2.set_ylim(0,1) # the axis for silhouette is [0,1]

fig.tight_layout()  # otherwise the right y-label is slightly clipped
```

```
plt.show()
```



2.1.1 5. Cluster with the optimal number

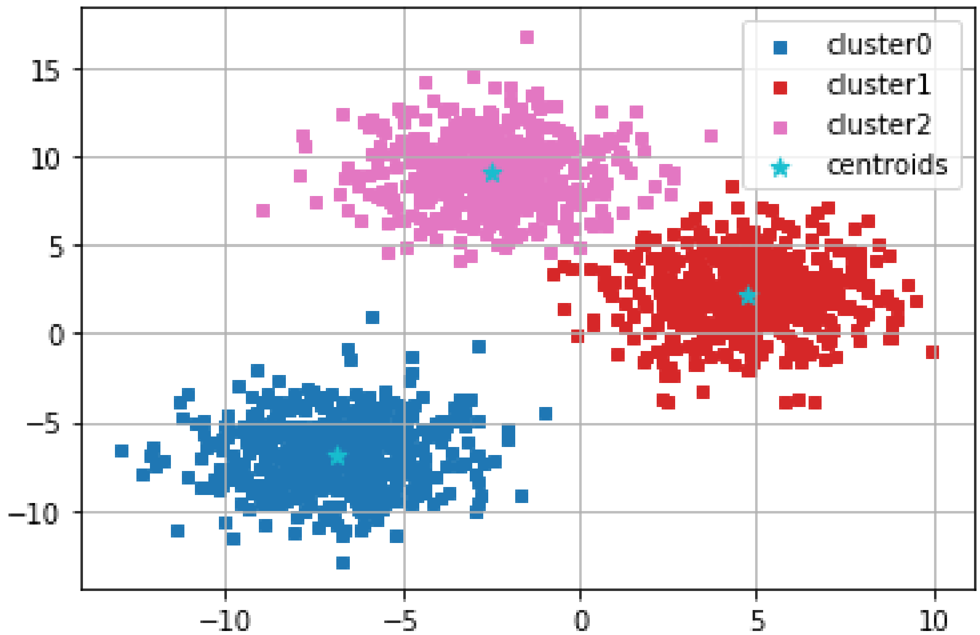
[11]:
```
good_k = 3
```

[12]:
```
km = KMeans(n_clusters=good_k,
            init='k-means++',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=rnd_state)
y_km = km.fit_predict(X)
```

[13]:
```
km.cluster_centers_
```

[13]:
```
array([[-6.89370123, -6.83658926,  0.52620605,  0.52371904],
       [ 4.75108211,  2.11850327,  0.4917521 ,  0.49502881],
       [-2.50474216,  9.09132188,  0.49394552,  0.48136246]])
```

[14]:
```
plot_clusters(X,y_km,dim=(focus[0],focus[1]), points = km.cluster_centers_)
```




```
[15]: print('Distortion: %.2f' % km.inertia_)
```

```
Distortion: 11831.85
```

2.2 Quantifying the quality of clustering via silhouette plots

The silhouette scores for the individual samples are computed with the function `silhouette_samples`

The function `plot_silhouette` produces a 'horizontal bar-plot', with one bar for each sample, where the length of the bar is proportional to the silhouette score of the sample. The bars are grouped for cluster and sorted for decreasing length.

A vertical line represents the silhouette score, i.e. the average on all the samples,

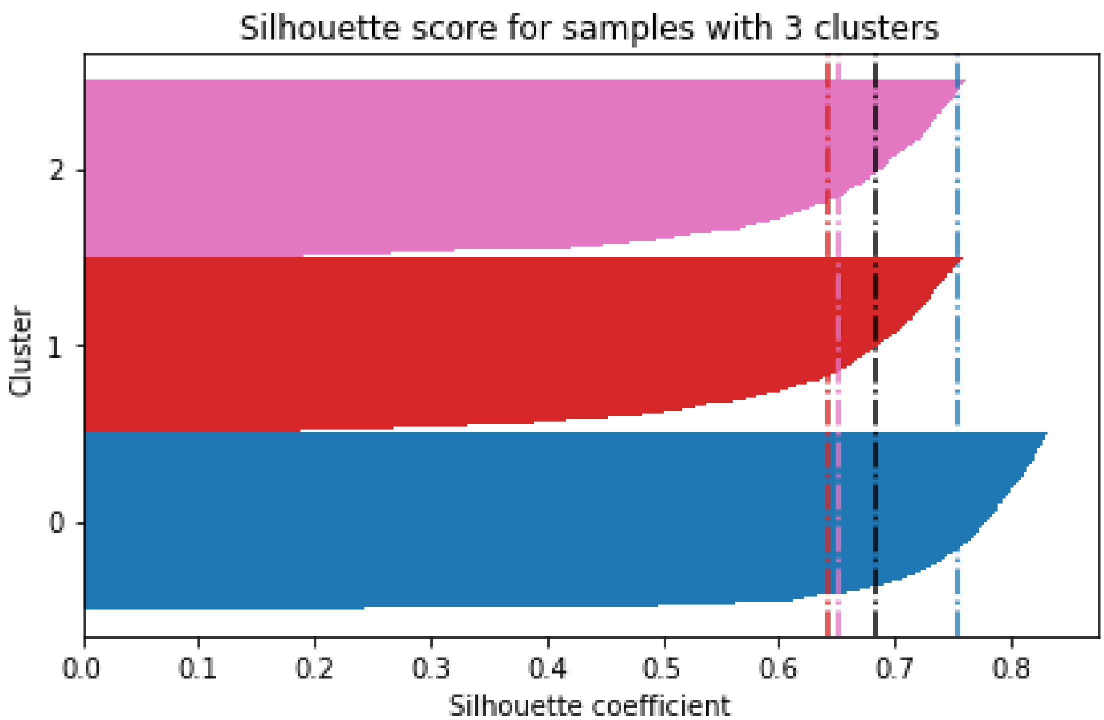
```
[16]: # from plot_silhouette import plot_silhouette
      from plot_silhouette2 import plot_silhouette
```

```
[17]: help(plot_silhouette)
```

```
Help on function plot_silhouette in module plot_silhouette2:

plot_silhouette(silhouette_vals, y, colors=<matplotlib.colors.ListedColormap
object at 0x11b0e5b50>, plot_noise=False)
    Plotting silhouette scores for the individual samples of a labelled data
set.
    The scores will be grouped according to labels and sorted in descending
```

```
order.
    The bars are proportional to the score and the color is determined by the
label.

    silhouette_vals: the silhouette values of the samples
    y:               the labels of the samples
    plot_noise:      boolean, assumes the noise to be labeled with a negative
integer
```

[18]:
```
cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0] # it is the number of rows
# Compute the Silhouette Coefficient for each sample, with the euclidean metric
silhouette_score_samples = silhouette_samples(X, y_km, metric='euclidean')
plt.title('Silhouette score for samples with {} clusters'.format(good_k))
plot_silhouette(silhouette_score_samples, y_km)
```

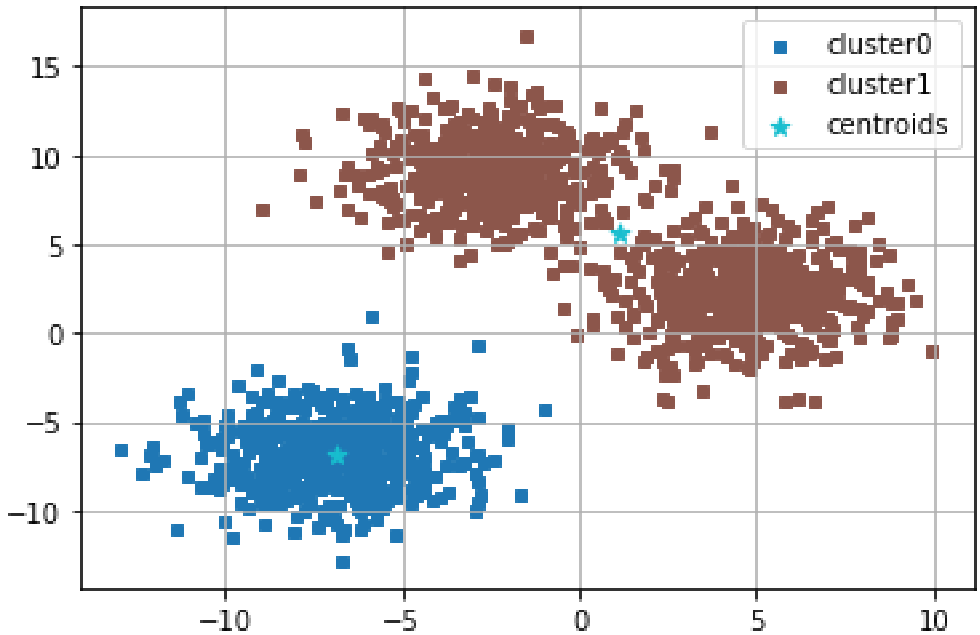


2.2.1 6. Comparison to "bad" clustering:

[19]:
```
bad_k = 2
```

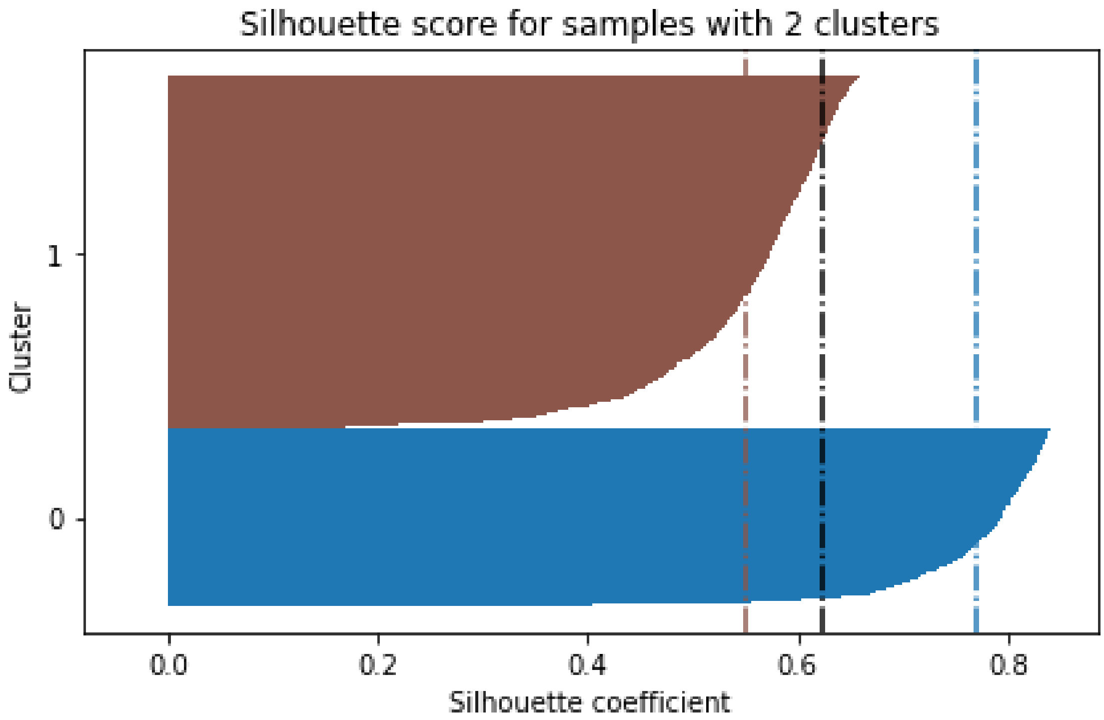
[20]:

```
km = KMeans(n_clusters=bad_k,
            init='k-means++',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=rnd_state)
y_km = km.fit_predict(X)
plot_clusters(X,y_km,dim=(focus[0],focus[1]), points = km.cluster_centers_)
```



[21]:

```
cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0]
silhouette_score_samples = silhouette_samples(X, y_km, metric='euclidean')
plt.title('Silhouette score for samples with {} clusters'.format(bad_k))
plot_silhouette(silhouette_score_samples, y_km)
```


Silhouette score for samples with 2 clusters
Cluster
1
0
0.0
0.2
0.4
0.6
0.8
Silhouette coefficient

```
[ ]:
```

ml-04-ex2-dbscan

February 12, 2020

Claudio Sartori

Elaboration from the example given in Sebastian Raschka, 2015

https://github.com/rasbt/python-machine-learning-book

1 Machine Learning - Lab

1.1 Working with Unlabeled Data – Clustering Analysis

1.1.1 Use DBSCAN

1.1.2 Overview

In this example we will use an *artificial* data set
1. load the data
2. check the shape and plot the content
3. observe the plot and decide which are the most interesting columns, to use in the plots of the clusters
- make a 2d plot of the two most promising columns
- use the 2d projection only for plotting, not for the other computations 4. initialize and `fit_predict` an estimator for `DBSCAN`, using the default parameters, then print the results - print the estimator to check the parameter values
- the labels are the unique values of the predicted values
- print if there is noise
- if there is noise the first cluster label will be `-1`
- print the number of clusters (noise excluded)
- the other clusters are labeled starting from `0`
- for each cluster (noise excluded) compute the **centroid**
- plot the data with the centroids and the colors representing clusters
- use the `plot_clusters` function provided
5. find the best parameters using `ParameterGrid` - prepare a dictionary with the parameters lists
- generate the list of the parameter combinations with `ParameterGrid` - for each combination of parameters
- initialize the DBSCAN estimator
- `fit_predict`
- extract the labels and the number of clusters excluding the *noise*
- compute the silhouette score and the number of unclustered objects (noise)

- filter and print the parameters and the results
- print if the silhouette score is above a threshold and the percentage of unclustered is below a threshold
6. observe visually the most promising combination of parameters
- fit and predict the estimator
- plot the clusters
- compute the silouette scores for the individual samples using the function `silhouette_samples`
- plot the silhouette scores for each sample using the function `plot_silhouette`

[1]:
```
from IPython.display import Image
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn.metrics import silhouette_score, silhouette_samples
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import ParameterGrid
from sklearn.preprocessing import MinMaxScaler

%matplotlib inline

rnd_state = 42 # This variable will be used in all the procedure calls allowing␣
↪a random_state parameter
               # in this way the running can be perfectly reproduced
               # just change this value for a different experiment

# the .py files with the functions provided must be in the same directory of␣
↪the .ipynb file
from plot_clusters import plot_clusters      # python script provided separately
from plot_silhouette import plot_silhouette  # python script provided separately
```

[2]:
```
help(plot_clusters)
```

```
Help on function plot_clusters in module plot_clusters:

plot_clusters(X, y, dim, points, labels_prefix='cluster',
points_name='centroids', colors=<matplotlib.colors.ListedColormap object at
0x11ebcb210>, points_color=(0.09019607843137255, 0.7450980392156863,
0.8117647058823529, 1.0))
    Plot a two dimensional projection of an array of labelled points
    X:      array with at least two columns
    y:      vector of labels, length as number of rows in X
    dim:    the two columns to project, inside range of X columns, e.g. (0,1)
    points: additional points to plot as 'stars'
    labels_prefix: prefix to the labels for the legend ['cluster']
    points_name:   legend name for the additional points ['centroids']
    colors: a color map
```

```
    points_color: the color for the points
```

[3]:
```python
help(plot_silhouette)
```

```
Help on function plot_silhouette in module plot_silhouette:

plot_silhouette(silhouette_vals, y, colors=<matplotlib.colors.ListedColormap
object at 0x11ebcb210>, plot_noise=False)
    Plotting silhouette scores for the individual samples of a labelled data
set.
    The scores will be grouped according to labels and sorted in descending
order.
    The bars are proportional to the score and the color is determined by the
label.

    silhouette_vals: the silhouette values of the samples
    y:               the labels of the samples
    plot_noise:      boolean, assumes the noise to be labeled with a negative
integer
```

1.1.3 1. Load the data

[4]:
```python
# data_file = 'ex1_4dim_data.csv'
# data_file = 'ex1_4dim_mod_data.csv'
# data_file = 'ex1_data.csv'
data_file = 'ex1_4d_moon.csv'
delimiter = ','
X = np.loadtxt(data_file, delimiter = delimiter)
# scaler = MinMaxScaler()
# X = scaler.fit_transform(X)
```

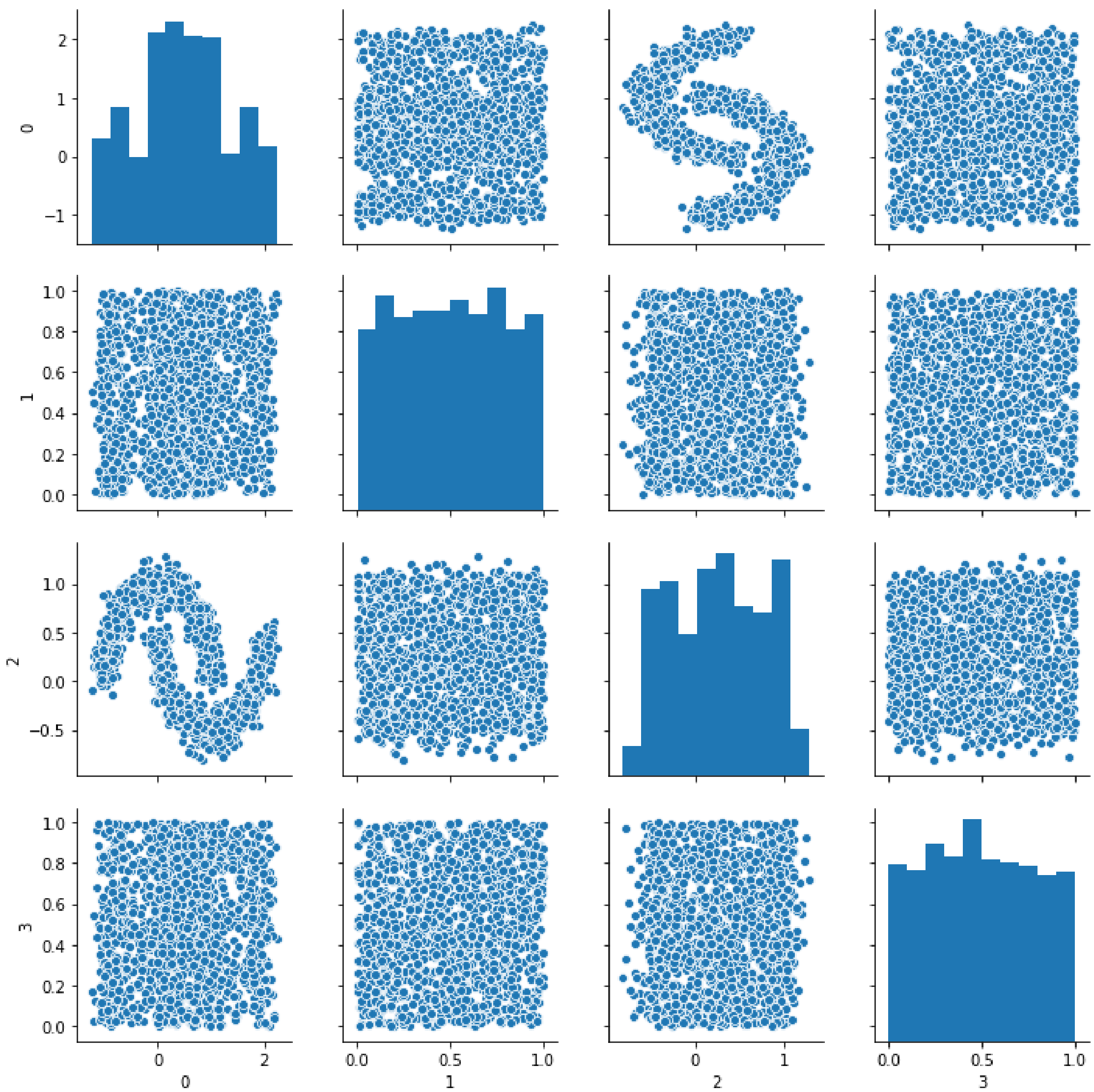
1.1.4 2. Inspect

[5]:
```python
X.shape
```

[5]: (1500, 4)

[6]:
```python
sns.pairplot(pd.DataFrame(X))
```

[6]: <seaborn.axisgrid.PairGrid at 0x1a2414bb90>



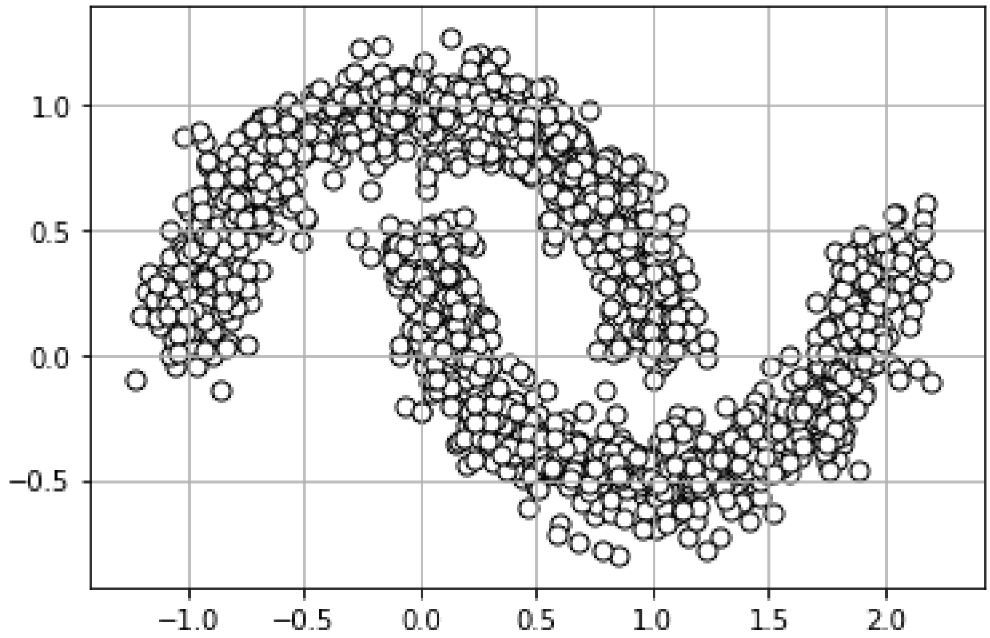
1.1.5 3. Observing the pairplots

In this simple example you can easily see which are the two most interesting columns.

All the plots will focus on those columns

[7]:
```
# focus = [0,1]
focus = [0,2]
plt.scatter(X[:,focus[0]], X[:,focus[1]]
            , c='white'           # color filling the data markers
            , edgecolors='black' # edge color for data markers
            , marker='o'          # data marker shape, e.g. triangles (v<>^),␣
 ↪square (s), star (*), ...
```

```
                , s=50)                 # data marker size
plt.grid()  # plots a grid on the data
plt.show()
```



1.1.6 4. Initialize, fit_predict and plot the clusters

[8]:
```
db = DBSCAN()
y_db = db.fit_predict(X)
```

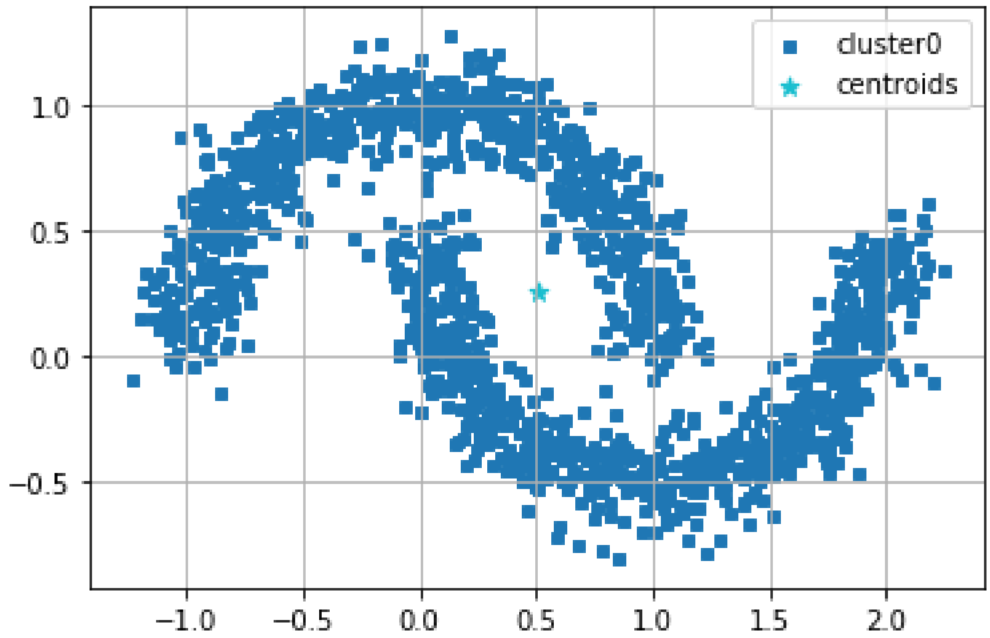
[9]:
```
print(db)
```

```
DBSCAN(algorithm='auto', eps=0.5, leaf_size=30, metric='euclidean',
       metric_params=None, min_samples=5, n_jobs=None, p=None)
```

[10]:
```
cluster_labels_all = np.unique(y_db)
cluster_labels = cluster_labels_all[cluster_labels_all != -1]
n_clusters = len(cluster_labels)
if cluster_labels_all[0] == -1:
    noise = True
    print("There is noise")
else:
    noise = False
print("There is/are {} cluster(s)".format(n_clusters-noise))
```

```
There is/are 1 cluster(s)
```

[11]:
```
cluster_centers = np.empty((n_clusters,X.shape[1]))
for i in cluster_labels:
    cluster_centers[i,:] = np.mean(X[y_db==i,:], axis = 0)
plot_clusters(X,y_db,dim=(focus[0],focus[1]), points = cluster_centers)
```



1.1.7 5. Find the best parameters using ParameterGrid

[12]:
```
param_grid = {'eps': list(np.arange(0.05, 1, 0.05)), 'min_samples':␣
↪list(range(1,10,1))}
params = list(ParameterGrid(param_grid))
sil_thr = 0  # visualize results only for combinations with silhouette above␣
↪the threshold
unc_thr = 33 # visualize results only for combinations with unclustered% below␣
↪the threshold
```

[13]:
```
print("{:11}\t{:11}\t{:11}\t{:11}\t{:11}".\
      format('        eps','min_samples',' n_clusters',' silhouette', '    ␣
↪unclust%'))
for i in range(len(params)):
    db = DBSCAN(**(params[i]))
    y_db = db.fit_predict(X)
    cluster_labels_all = np.unique(y_db)
    cluster_labels = cluster_labels_all[cluster_labels_all != -1]
    n_clusters = len(cluster_labels)
```

```
    if n_clusters > 1:
        X_cl = X[y_db!=-1,:]
        y_db_cl = y_db[y_db!=-1]
        silhouette = silhouette_score(X_cl,y_db_cl)
        uncl_p = (1 - y_db_cl.shape[0]/y_db.shape[0]) * 100
        if silhouette > sil_thr and uncl_p < unc_thr:
            print("{:11.2f}\t{:11}\t{:11}\t{:11.2f}\t{:11.2f}%"\
                  .format(db.eps, db.min_samples, n_clusters, silhouette,␣
↪uncl_p))
```

```
        eps     min_samples      n_clusters      silhouette        unclust%
       0.05               1            1490            0.01           0.00%
       0.10               1            1297            0.09           0.00%
       0.15               1             698            0.07           0.00%
       0.15               2             253            0.25          29.67%
       0.25               2               2            0.14           1.27%
       0.25               7               2            0.28           3.27%
       0.25               8               2            0.28           5.80%
       0.25               9               2            0.29           8.47%
```

1.1.8 6. Observe

- Observe visually the most promising combination of parameters.
- Plot the clusters with the centers
- Plot the silhouette indexes for all the clustered samples

[14]:
```
# db = DBSCAN(eps=0.9, min_samples=4)
db = DBSCAN(eps=0.25, min_samples=9)
y_db = db.fit_predict(X)
cluster_labels_all = np.unique(y_db)
cluster_labels = cluster_labels_all[cluster_labels_all != -1]
n_clusters = len(cluster_labels)
```

[15]:
```
cluster_centers = np.empty((n_clusters,X.shape[1]))
for i in cluster_labels:
    cluster_centers[i,:] = np.mean(X[y_db==i,:], axis = 0)
```

[16]:
```
print("There are {} clusters".format(n_clusters))
```

```
There are 2 clusters
```

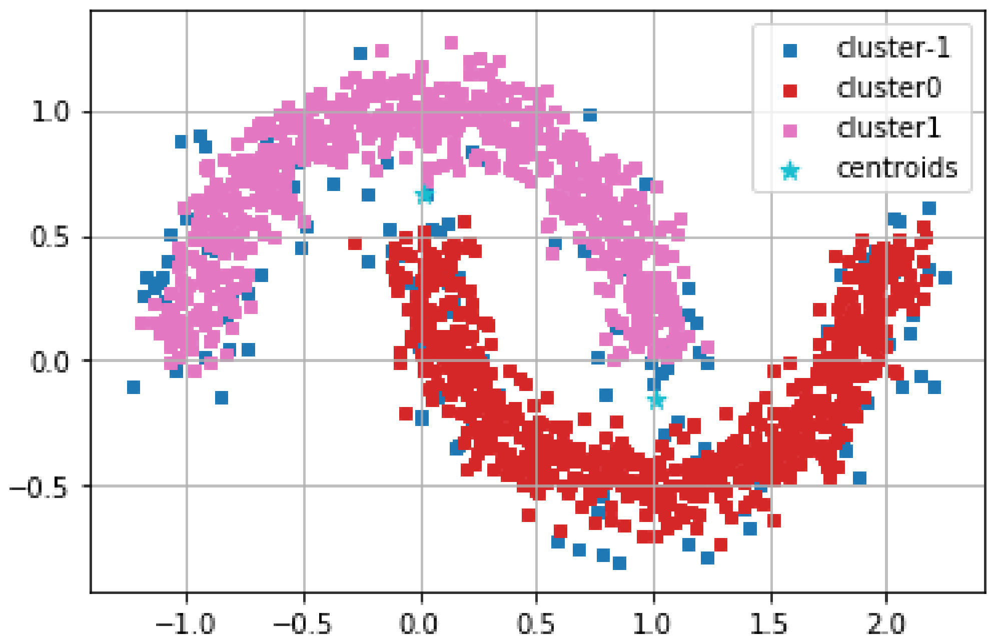
[17]:
```
print("The cluster labels are {}".format(cluster_labels))
```

```
The cluster labels are [0 1]
```

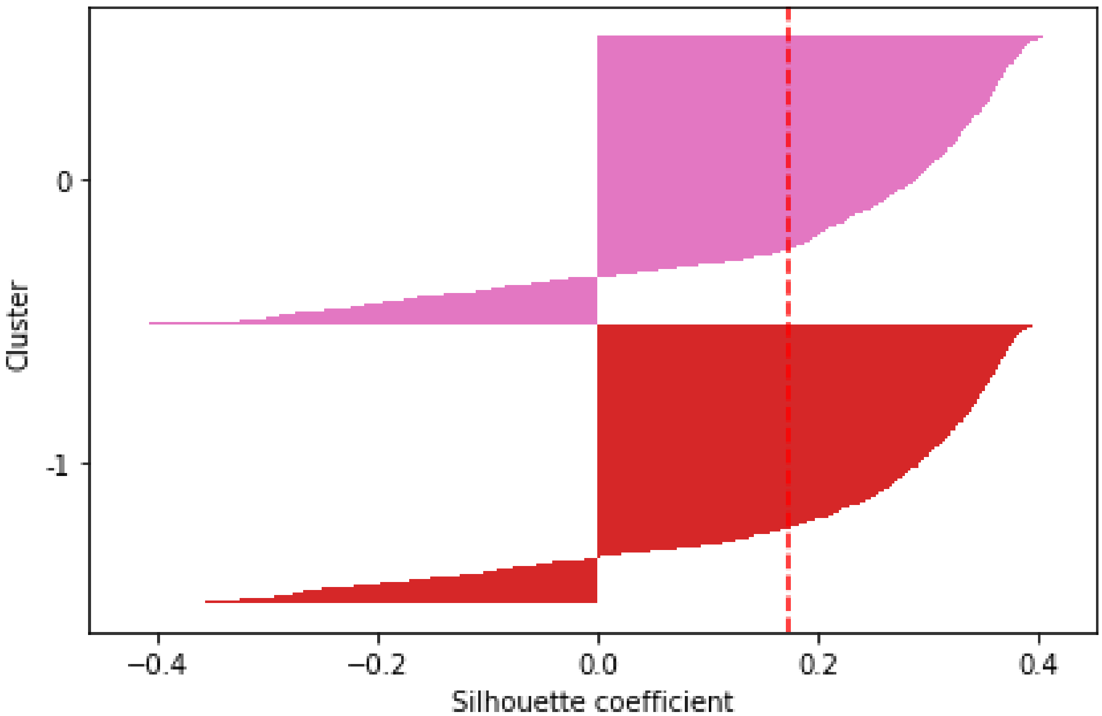
[18]:
```
cluster_centers
```

[18]:
```
array([[ 1.01020828,  0.5258109 , -0.15197393,  0.49321332],
       [ 0.01924021,  0.48326285,  0.66582183,  0.4831518 ]])
```

[19]:
```
plot_clusters(X,y_db,dim=(focus[0],focus[1]), points = cluster_centers)
```



[20]:
```
# X_cl = X[y_db!=-1,:]
# y_db_cl = y_db[y_db!=-1]
# silhouette = silhouette_samples(X_cl,y_db_cl)
# plot_silhouette(silhouette,y_db_cl)
silhouette = silhouette_samples(X,y_db)
plot_silhouette(silhouette,y_db)
```


0
-1
Cluster
−0.4
−0.2
0.0
0.2
0.4
Silhouette coefficient

ml-05-association-rules

February 12, 2020

1 Association Rules

1.1 Example with the Online Retail dataset, from UCI

Code provided in this link

[1]:
```
import pandas as pd
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
```

Upload the file `'Online-Retail.xlsx'`. It is a MS Excel file, you can read it with the Pandas' function `read_excel`.

Inspect its content. It is a transactional database where the role of transaction identifier is played by the column `InvoiceNo` and the items are in the column `Description`.

The database has some problems: 1. some descriptions represent the same item but have different leading or trailing spaces, therefore they must be made uniform with the Pandas' function `str.strip()`

[2]:
```
# url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00352/
↪Online%20Retail.xlsx'
# url = 'machineLearning-05-association-rules-lab/Online-Retail.xlsx'
# url = 'Online-Retail.xlsx'
# df0 = pd.read_excel(url)
url = 'Online-Retail.csv'
df0 = pd.read_csv(url)
df0.head(20)
```

[2]:
```
  InvoiceNo StockCode                          Description  Quantity  \
0    536365    85123A   WHITE HANGING HEART T-LIGHT HOLDER         6
1    536365     71053                  WHITE METAL LANTERN         6
2    536365    84406B       CREAM CUPID HEARTS COAT HANGER         8
3    536365    84029G  KNITTED UNION FLAG HOT WATER BOTTLE         6
4    536365    84029E       RED WOOLLY HOTTIE WHITE HEART.         6
5    536365     22752         SET 7 BABUSHKA NESTING BOXES         2
6    536365     21730    GLASS STAR FROSTED T-LIGHT HOLDER         6
7    536366     22633               HAND WARMER UNION JACK         6
8    536366     22632            HAND WARMER RED POLKA DOT         6
```

```
9         NaN     84879         ASSORTED COLOUR BIRD ORNAMENT        32
10     536367     22745            POPPY'S PLAYHOUSE BEDROOM          6
11     536367     22748             POPPY'S PLAYHOUSE KITCHEN         6
12     536367     22749     FELTCRAFT PRINCESS CHARLOTTE DOLL         8
13     536367     22310                IVORY KNITTED MUG COSY         6
14     536367     84969    BOX OF 6 ASSORTED COLOUR TEASPOONS         6
15     536367     22623         BOX OF VINTAGE JIGSAW BLOCKS          3
16     536367     22622        BOX OF VINTAGE ALPHABET BLOCKS         2
17     536367     21754              HOME BUILDING BLOCK WORD         3
18     536367     21755              LOVE BUILDING BLOCK WORD         3
19     536367     21777           RECIPE BOX WITH METAL HEART         4

            InvoiceDate  UnitPrice  CustomerID         Country
0   2010-12-01 08:26:00       2.55     17850.0  United Kingdom
1   2010-12-01 08:26:00       3.39     17850.0  United Kingdom
2   2010-12-01 08:26:00       2.75     17850.0  United Kingdom
3   2010-12-01 08:26:00       3.39     17850.0  United Kingdom
4   2010-12-01 08:26:00       3.39     17850.0  United Kingdom
5   2010-12-01 08:26:00       7.65     17850.0  United Kingdom
6   2010-12-01 08:26:00       4.25     17850.0  United Kingdom
7   2010-12-01 08:28:00       1.85     17850.0  United Kingdom
8   2010-12-01 08:28:00       1.85     17850.0  United Kingdom
9   2010-12-01 08:34:00       1.69     13047.0  United Kingdom
10  2010-12-01 08:34:00       2.10     13047.0  United Kingdom
11  2010-12-01 08:34:00       2.10     13047.0  United Kingdom
12  2010-12-01 08:34:00       3.75     13047.0  United Kingdom
13  2010-12-01 08:34:00       1.65     13047.0  United Kingdom
14  2010-12-01 08:34:00       4.25     13047.0  United Kingdom
15  2010-12-01 08:34:00       4.95     13047.0  United Kingdom
16  2010-12-01 08:34:00       9.95     13047.0  United Kingdom
17  2010-12-01 08:34:00       5.95     13047.0  United Kingdom
18  2010-12-01 08:34:00       5.95     13047.0  United Kingdom
19  2010-12-01 08:34:00       7.95     13047.0  United Kingdom
```

[3]:
```
url_csv = 'Online-Retail.csv'
df0.to_csv(url_csv, index=False)
```

[4]:
```
df0.head()
```

[4]:
```
  InvoiceNo StockCode                          Description  Quantity  \
0    536365    85123A   WHITE HANGING HEART T-LIGHT HOLDER         6
1    536365     71053                  WHITE METAL LANTERN         6
2    536365    84406B       CREAM CUPID HEARTS COAT HANGER         8
3    536365    84029G  KNITTED UNION FLAG HOT WATER BOTTLE         6
4    536365    84029E       RED WOOLLY HOTTIE WHITE HEART.         6

           InvoiceDate  UnitPrice  CustomerID         Country
```

```
0  2010-12-01 08:26:00       2.55     17850.0  United Kingdom
1  2010-12-01 08:26:00       3.39     17850.0  United Kingdom
2  2010-12-01 08:26:00       2.75     17850.0  United Kingdom
3  2010-12-01 08:26:00       3.39     17850.0  United Kingdom
4  2010-12-01 08:26:00       3.39     17850.0  United Kingdom
```

[5]:
```python
print("The number of unique Description values in the input file is {}".
 ↪format(len(df0['Description'].unique())))
```

```
The number of unique Description values in the input file is 4224
```

[6]:
```python
df1 = df0
df1['Description'] = df0['Description'].str.strip()
```

[7]:
```python
print("After cleaning, the number of unique Description values in the input␣
 ↪file is {}".format(len(df1['Description'].unique())))
```

```
After cleaning, the number of unique Description values in the input file is
4212
```

Some rows may not have an `InvoiceNo` and must be removed, because they cannot be used.

Check if there are such that rows and in case remove them. You can check with the Pandas' function `isna` and remove with `dropna` on `axis=0`, with the option `subset`

[8]:
```python
print("Rows with missing InvoiceNo before removing")
df1[df1['InvoiceNo'].isna()]
```

```
Rows with missing InvoiceNo before removing
```

[8]:
```
  InvoiceNo StockCode                     Description  Quantity  \
9       NaN     84879  ASSORTED COLOUR BIRD ORNAMENT        32

           InvoiceDate  UnitPrice  CustomerID         Country
9  2010-12-01 08:34:00       1.69     13047.0  United Kingdom
```

[9]:
```python
df2 = df1.dropna(axis=0, subset=['InvoiceNo'])
```

[10]:
```python
print("Rows with missing InvoiceNo after removing")
df2[df2['InvoiceNo'].isna()]
```

```
Rows with missing InvoiceNo after removing
```

[10]:
```
Empty DataFrame
Columns: [InvoiceNo, StockCode, Description, Quantity, InvoiceDate, UnitPrice,
CustomerID, Country]
Index: []
```

Some InvoiceNo start with a `C`. They are ”credit transactions” and must be removed.

Check the number of rows containing `C` in `InvoiceNo` and remove them. At the moment the column `InvoiceNo` is a generic object, in order to be able to use string functions, such as `contains`, it must be transformed into `str` with `astype`.

[11]:
```
print("There are {} rows containing 'C' in 'InvoiceNo'"\
      .format(sum(df2['InvoiceNo'].astype('str').str.contains('C'))))
```

```
There are 9288 rows containing 'C' in 'InvoiceNo'
```

[12]:
```
df3 = df2[~df2['InvoiceNo'].astype('str').str.contains('C')]
```

[13]:
```
print("After removal, there are {} rows containing 'C' in 'InvoiceNo'"\
      .format(sum(df3['InvoiceNo'].astype('str').str.contains('C'))))
```

```
After removal, there are 0 rows containing 'C' in 'InvoiceNo'
```

Several transactions include the item `'POSTAGE'`, which represents the mailing expenses. In this analysis we are not interested in it, therefore the rows with `'POSTAGE'` will be removed.

[14]:
```
container = 'Description'
target = 'POSTAGE'
print("There are {} rows containing {} in {}"\
      .format(sum(df2[container].astype('str').str.contains(target)), target,␣
 ↪container))
```

```
There are 1961 rows containing POSTAGE in Description
```

[15]:
```
df = df3[~df3[container].astype('str').str.contains(target)]
```

[16]:
```
df.describe
```

[16]:
```
<bound method NDFrame.describe of        InvoiceNo StockCode
Description  Quantity  \
0         536365    85123A   WHITE HANGING HEART T-LIGHT HOLDER         6
1         536365     71053                  WHITE METAL LANTERN         6
2         536365    84406B       CREAM CUPID HEARTS COAT HANGER         8
3         536365    84029G  KNITTED UNION FLAG HOT WATER BOTTLE         6
4         536365    84029E       RED WOOLLY HOTTIE WHITE HEART.         6
…            …        …                                  …        …
541904    581587     22613          PACK OF 20 SPACEBOY NAPKINS        12
541905    581587     22899         CHILDREN'S APRON DOLLY GIRL          6
541906    581587     23254        CHILDRENS CUTLERY DOLLY GIRL          4
541907    581587     23255     CHILDRENS CUTLERY CIRCUS PARADE          4
541908    581587     22138        BAKING SET 9 PIECE RETROSPOT          3

                InvoiceDate  UnitPrice  CustomerID         Country
0       2010-12-01 08:26:00       2.55     17850.0  United Kingdom
1       2010-12-01 08:26:00       3.39     17850.0  United Kingdom
2       2010-12-01 08:26:00       2.75     17850.0  United Kingdom
```

```
3       2010-12-01 08:26:00       3.39     17850.0  United Kingdom
4       2010-12-01 08:26:00       3.39     17850.0  United Kingdom
…                         …        …         …            …
541904  2011-12-09 12:50:00       0.85     12680.0          France
541905  2011-12-09 12:50:00       2.10     12680.0          France
541906  2011-12-09 12:50:00       4.15     12680.0          France
541907  2011-12-09 12:50:00       4.15     12680.0          France
541908  2011-12-09 12:50:00       4.95     12680.0          France

[530786 rows x 8 columns]>
```

After the cleanup, we need to consolidate the items into 1 transaction per row with each product 1 hot encoded. For the sake of keeping the data set small, we are only looking at sales for France. However, in additional code below, we will compare these results to sales from Germany. Further country comparisons would be interesting to investigate.

Actions: 1. filter the rows 'Country='France'2. group by['InvoiceNo', 'Description']computing a sum on['Quantity']3. use theunstackfunction to move the items from rows to columns 4. reset the index 5. fill the missing with zero (fillna(0)) 6. store the result in the new dataframebasket' and inspect it

[17]:
```python
basket = (df[df['Country'] =="France"]
          .groupby(['InvoiceNo', 'Description'])['Quantity']
          .sum().unstack().reset_index().fillna(0)
          .set_index('InvoiceNo'))
basket.head()
```

[17]:
```
Description  10 COLOUR SPACEBOY PEN  12 COLOURED PARTY BALLOONS  \
InvoiceNo
536370                          0.0                         0.0
536852                          0.0                         0.0
536974                          0.0                         0.0
537065                          0.0                         0.0
537463                          0.0                         0.0

Description  12 EGG HOUSE PAINTED WOOD  12 MESSAGE CARDS WITH ENVELOPES  \
InvoiceNo
536370                             0.0                              0.0
536852                             0.0                              0.0
536974                             0.0                              0.0
537065                             0.0                              0.0
537463                             0.0                              0.0

Description  12 PENCIL SMALL TUBE WOODLAND  \
InvoiceNo
536370                                 0.0
536852                                 0.0
536974                                 0.0
```

```
537065                                0.0
537463                                0.0

Description  12 PENCILS SMALL TUBE RED RETROSPOT  12 PENCILS SMALL TUBE SKULL  \
InvoiceNo
536370                                      0.0                           0.0
536852                                      0.0                           0.0
536974                                      0.0                           0.0
537065                                      0.0                           0.0
537463                                      0.0                           0.0

Description  12 PENCILS TALL TUBE POSY  12 PENCILS TALL TUBE RED RETROSPOT  \
InvoiceNo
536370                             0.0                                 0.0
536852                             0.0                                 0.0
536974                             0.0                                 0.0
537065                             0.0                                 0.0
537463                             0.0                                 0.0

Description  12 PENCILS TALL TUBE WOODLAND  …  WRAP VINTAGE PETALS  DESIGN  \
InvoiceNo                                   …
536370                                 0.0  …                         0.0
536852                                 0.0  …                         0.0
536974                                 0.0  …                         0.0
537065                                 0.0  …                         0.0
537463                                 0.0  …                         0.0

Description  YELLOW COAT RACK PARIS FASHION  YELLOW GIANT GARDEN THERMOMETER  \
InvoiceNo
536370                                  0.0                              0.0
536852                                  0.0                              0.0
536974                                  0.0                              0.0
537065                                  0.0                              0.0
537463                                  0.0                              0.0

Description  YELLOW SHARK HELICOPTER  ZINC  STAR T-LIGHT HOLDER  \
InvoiceNo
536370                           0.0                        0.0
536852                           0.0                        0.0
536974                           0.0                        0.0
537065                           0.0                        0.0
537463                           0.0                        0.0

Description  ZINC FOLKART SLEIGH BELLS  ZINC HERB GARDEN CONTAINER  \
InvoiceNo
536370                             0.0                         0.0
536852                             0.0                         0.0
```

```
536974                             0.0                         0.0
537065                             0.0                         0.0
537463                             0.0                         0.0

Description  ZINC METAL HEART DECORATION  ZINC T-LIGHT HOLDER STAR LARGE  \
InvoiceNo
536370                               0.0                             0.0
536852                               0.0                             0.0
536974                               0.0                             0.0
537065                               0.0                             0.0
537463                               0.0                             0.0

Description  ZINC T-LIGHT HOLDER STARS SMALL
InvoiceNo
536370                                   0.0
536852                                   0.0
536974                                   0.0
537065                                   0.0
537463                                   0.0

[5 rows x 1562 columns]
```

There are a lot of zeros in the data but we also need to make sure any positive values are converted to a 1 and anything less the 0 is set to 0.

You can define a function `encode_units` which takes a number and returns 0 if the number is 0 or less, 1 if the number is 1 or more. The function can be applied to `basket` with the Pandas' function `applymap`, the result is stored in the variable `basket_sets`

This step will complete the one hot encoding of the data.

[18]:
```
import matplotlib.pyplot as plt
%matplotlib inline
def encode_units(x):
    if x <= 0:
        return 0
    if x >= 1:
        return 1

basket_sets = basket.applymap(encode_units)
```

Now that the data is structured properly, we can generate frequent item sets that have a support of at least 7% (this number was chosen so that we can get enough useful examples):

- generate the `frequent_itemsets` with `apriori`, setting `min_support=0.07` and `use_colnames=True`
- generate the `rules` with `association_rules` using `metric="lift"` and `min_threshold=1`
- show the rules

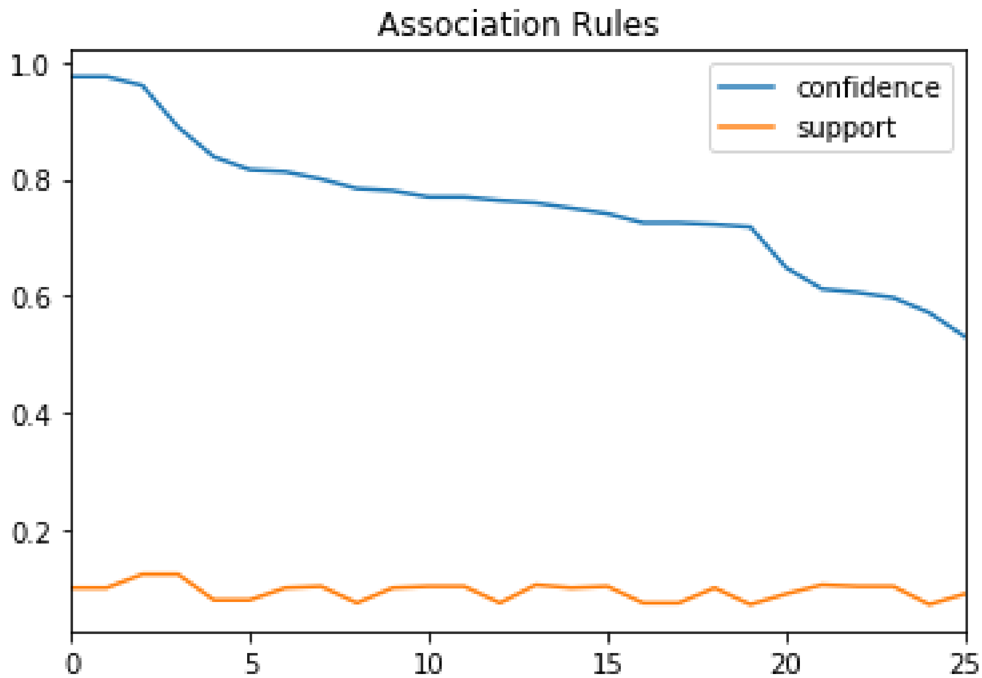
[19]:
```
frequent_itemsets = apriori(basket_sets, min_support=0.07, use_colnames=True)

rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1)
rules.head()
```

[19]:
```
                    antecedents                    consequents  \
0  (ALARM CLOCK BAKELIKE GREEN)   (ALARM CLOCK BAKELIKE PINK)
1   (ALARM CLOCK BAKELIKE PINK)  (ALARM CLOCK BAKELIKE GREEN)
2  (ALARM CLOCK BAKELIKE GREEN)    (ALARM CLOCK BAKELIKE RED)
3    (ALARM CLOCK BAKELIKE RED)  (ALARM CLOCK BAKELIKE GREEN)
4    (ALARM CLOCK BAKELIKE RED)   (ALARM CLOCK BAKELIKE PINK)

   antecedent support  consequent support   support  confidence      lift  \
0            0.098191            0.103359  0.074935    0.763158  7.383553
1            0.103359            0.098191  0.074935    0.725000  7.383553
2            0.098191            0.095607  0.080103    0.815789  8.532717
3            0.095607            0.098191  0.080103    0.837838  8.532717
4            0.095607            0.103359  0.074935    0.783784  7.583108

   leverage  conviction
0  0.064786    3.785817
1  0.064786    3.279305
2  0.070716    4.909561
3  0.070716    5.561154
4  0.065054    4.146964
```

In order to plot the rules, it is better to sort them according to some metrics. We will sort on descending confidence and support and plot `'confidence'` and `'support'`.

[20]:
```
sorted_rules=rules.sort_values(by=['confidence','support'],ascending=False).
↪reset_index(drop=True)
sorted_rules.loc[:,['confidence','support']].plot(title='Association Rules')
```

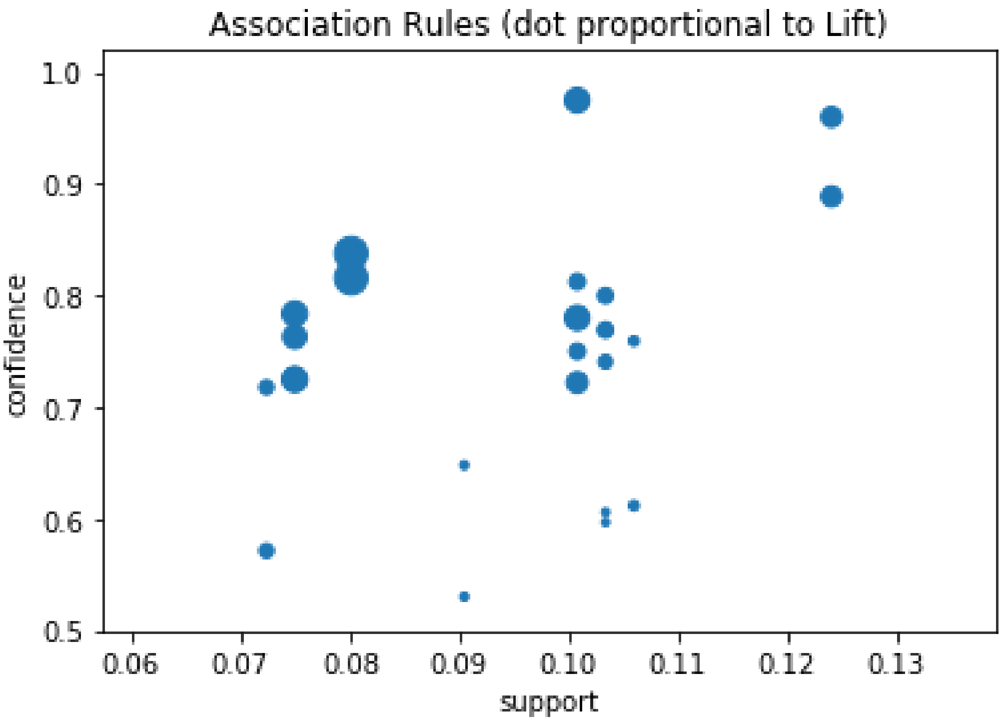
[20]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11eeaa610>
```



You find below a three dimensional plot, where the dot size is proportional to the lift, obtained using `plot.scatter`.

[21]:
```
s = [1.8**n for n in rules.lift]
rules.plot.scatter(x='support',
                   y='confidence',
                   title='Association Rules (dot proportional to Lift)',
                   s=s)
```

[21]: `<matplotlib.axes._subplots.AxesSubplot at 0x11ef2c610>`



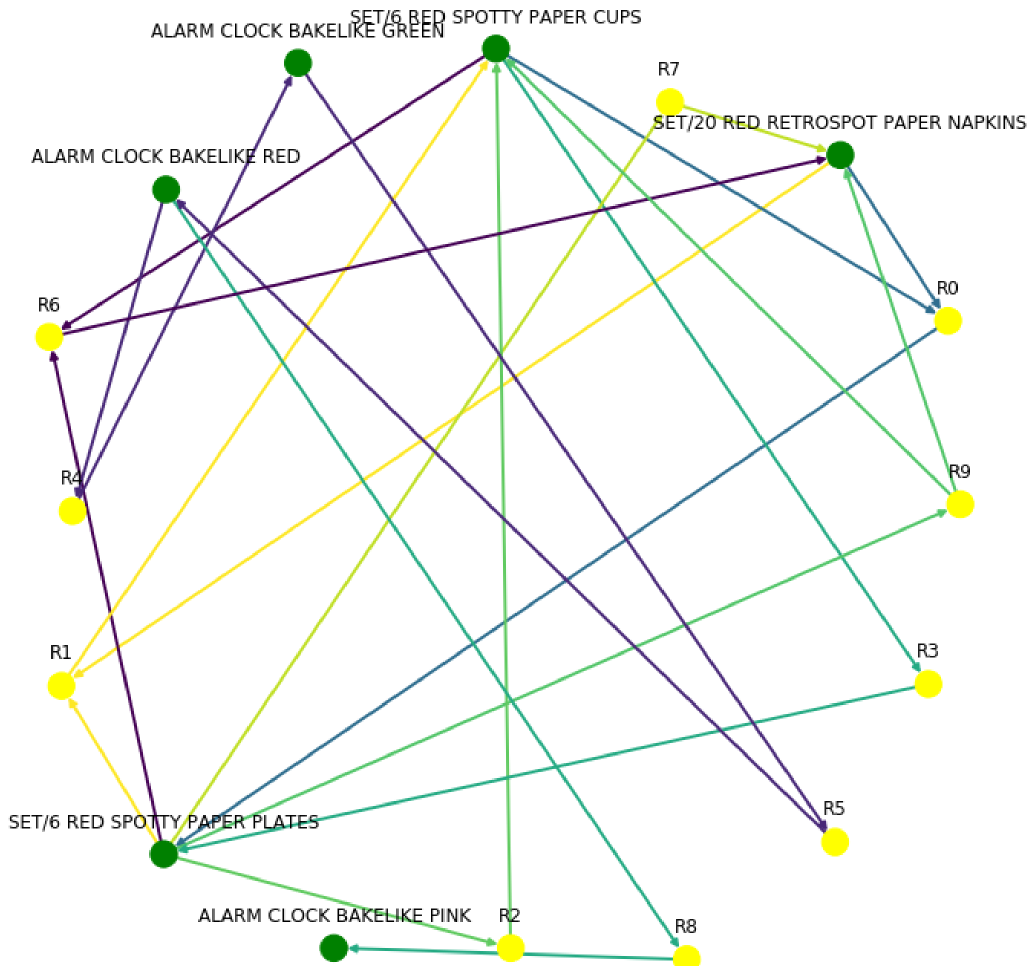
Finally, we draw a plot of a subset of the rules using the function `draw_graph`, provided in this package.

[23]:
```
from draw_rules_graph import draw_graph
help(draw_graph)
```

```
Help on function draw_graph in module draw_rules_graph:

draw_graph(rules, rules_to_show=5)
    draws the rules as a graph linking antecedents and consequents
    "rule nodes" are yellow, with name "R<n>", "item nodes" are green
    arrows colors are different for each rule, and go from the antecedent(s)
    to the rule node and to the consequent(s)
    the "rules_to_show" parameter limits the rules to show to the initial
    part of the "rules" dataframe
```

[24]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.gcf().clear()
plt.figure(figsize=(10,10))
draw_graph (sorted_rules, 10)
```

```
<Figure size 432x288 with 0 Axes>
```




```
[ ]:
```

ml-lab17-12-2019-dbscan

February 12, 2020

Claudio Sartori©

1 Machine Learning - Lab

1.1 Example of Lab exam

1.1.1 Find the best clustering with DBSCAN

2 Tasks

Find the clusters in the included dataset.

The solution must be produced as a Python Notebook. The notebook must include appropriate comments and must produce:

1. the boxplots of the attributes and a comment on remarkable situations, if any (2pt)
2. a pairplot of the data (see Seaborn pairplot) and a comment on remarkable situations, if any (2pt)
3. a clustering schema using a method of your choice exploring a range of parameter values (5pt)
4. the plot of the global inertia (SSD) and silhouette index for the parameter values you examine (4pt)
5. the optimal parameters of your choice (4pt)
6. a pairplot of the data using as hue the cluster assignment with the optimal parameter (3pt)
7. a plot of the silhouette index for the data points, grouped according to the clusters (4pt)
8. A sorted list of the discovered clusters for decreasing sizes (7pt)

NB: this solution is much more than what was requested for the exam

[1]:
```
from IPython.display import Image
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn.metrics import silhouette_score, silhouette_samples
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import ParameterGrid
```

```
from sklearn.preprocessing import MinMaxScaler

%matplotlib inline

rnd_state = 42 # This variable will be used in all the procedure calls allowing␣
 ↪a random_state parameter
               # in this way the running can be perfectly reproduced
               # just change this value for a different experiment

# the .py files with the functions provided must be in the same directory of␣
 ↪the .ipynb file
from plot_clusters import plot_clusters      # python script provided separately
from plot_silhouette import plot_silhouette  # python script provided separately
```

[2]:
```
# data_file = 'ex1_4dim_data.csv'
# data_file = 'ex1_4dim_mod_data.csv'
# data_file = 'ex1_data.csv'
data_file = 'lab_exercise.csv'
delimiter = ','
X = np.loadtxt(data_file, delimiter = delimiter)
# scaler = MinMaxScaler()
# X = scaler.fit_transform(X)
```

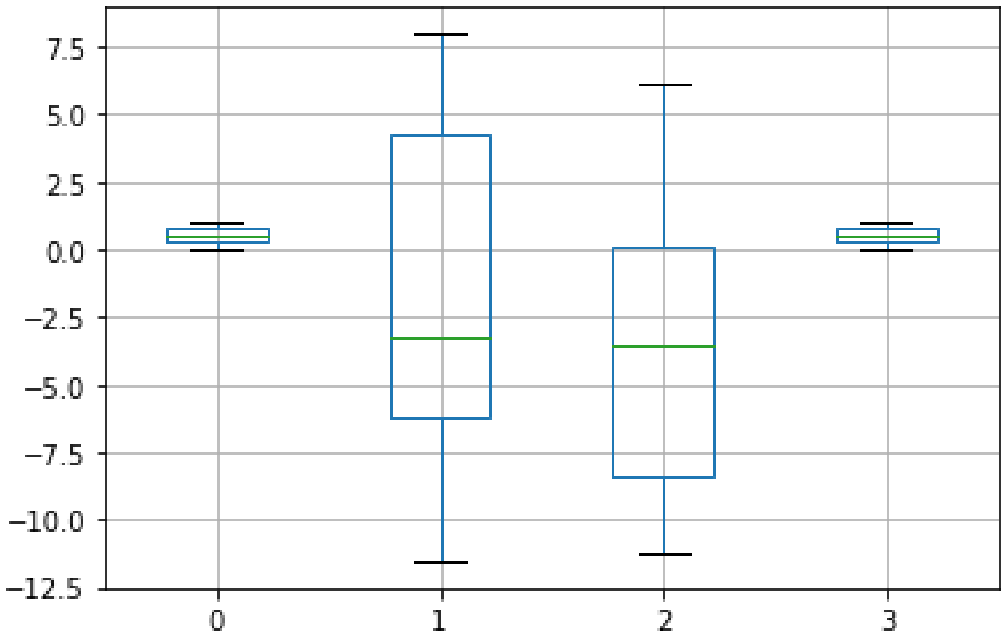
[3]:
```
X.shape
```

[3]:
```
(1500, 4)
```

In order to exploit the Pandas DataFrame features we generate `df` from `X`

[4]:
```
df = pd.DataFrame(X)
df.boxplot()
```

[4]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1084bea50>
```



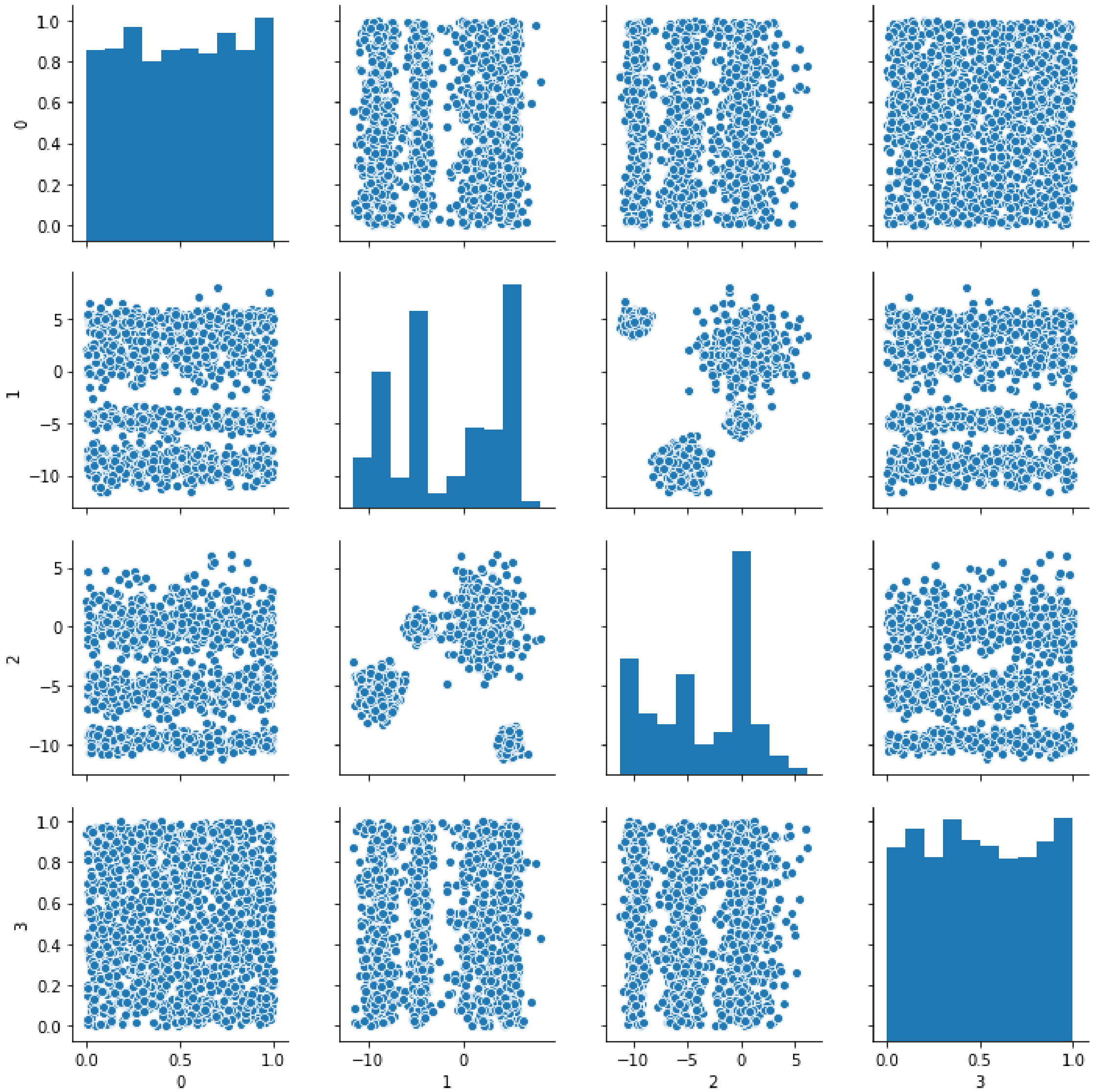
2.1 1. Comment on boxplots

Columns 0 and 3 have a range much smaller than 1 and 2. The distributions onf 0 and 3 seem to be equal. Poddibly, a min-max rescaling could point out some additional insight.

Let's look at the pairplots and consider if it is worth to do this transformation.

[5]:
```
sns.pairplot(pd.DataFrame(X))
```

[5]: <seaborn.axisgrid.PairGrid at 0x1a1cd1fb50>



2.2 2. Comments on pairplots

The pairplots show that the two most interesting columns are `1` and `2`, their pairplot shows evident clusters.

The pairplots of `0` and `3` show that those columns are uniformly distributed and do not show any pattern.

[6]:
```
int_cols = [1,2] # Interesting columns
```

2.3 3. Find a clustering scheme with DBSCAN

We will try **k_means** with a number of clusters varying from 2 to 10

Fit a DBSCAN estimator with the default parameters and examine the results.

[7]:
```
db = DBSCAN()
y_db = db.fit_predict(X)
```

[8]:
```
print(db)
```

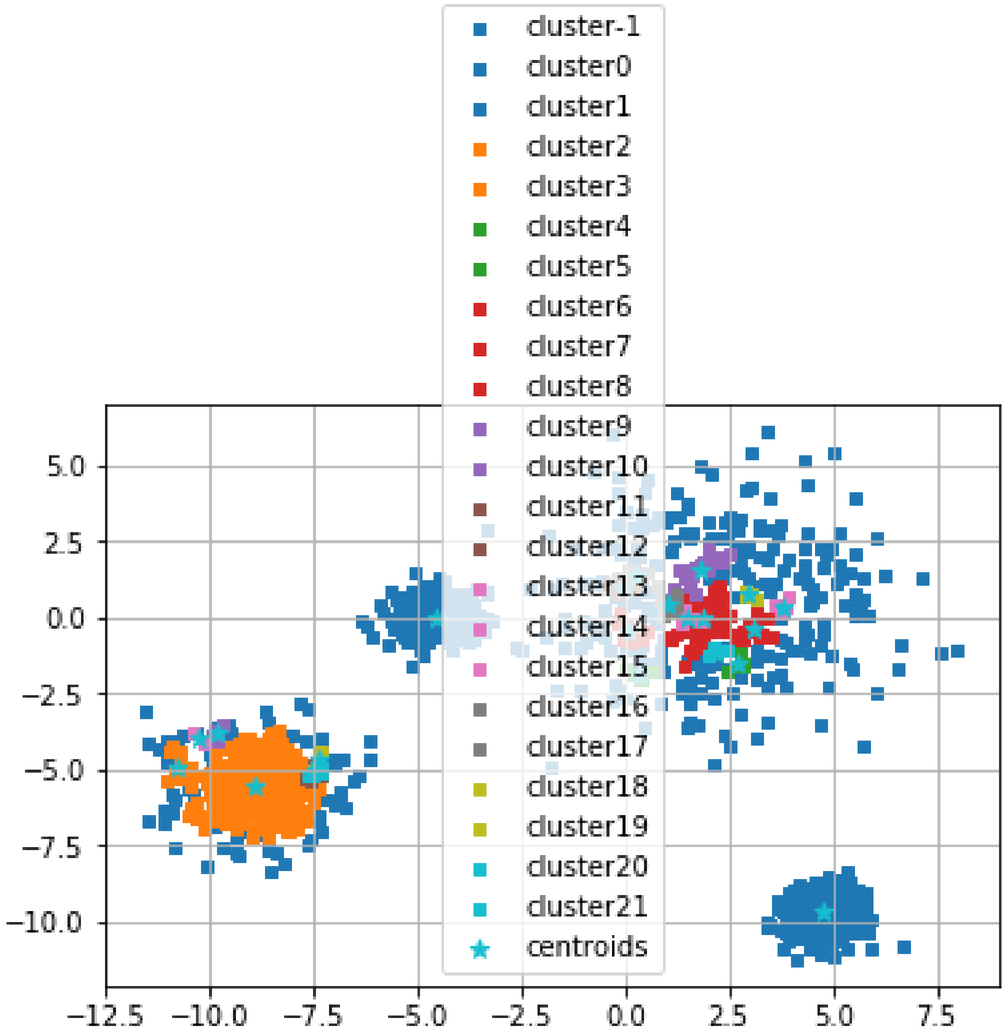
```
DBSCAN(algorithm='auto', eps=0.5, leaf_size=30, metric='euclidean',
       metric_params=None, min_samples=5, n_jobs=None, p=None)
```

[9]:
```
# Extract the unique labels
cluster_labels_all = np.unique(y_db)
# Exclude -1 (the label of noise) from the unique labels
cluster_labels = cluster_labels_all[cluster_labels_all != -1]
n_clusters = len(cluster_labels)
# If there is noise the label -1 will be the first one,
#    according to the documentation
if cluster_labels_all[0] == -1:
    noise = True
    print("There is noise")
else:
    noise = False
print("There is/are {} cluster(s)".format(n_clusters-noise))
```

```
There is noise
There is/are 21 cluster(s)
```

Plot the clusters and their centers according to the clustering schema obtained with the default parameters. Plot only the dimensions which have been considered interesting.

[10]:
```
# Prepare a data structure for the centroids
# the number of components is extracted from the shape of X
cluster_centers = np.empty((n_clusters,X.shape[1]))

# for each cluster label filter the data and compute the centroids
# as the mean along each component
for i in cluster_labels:
    cluster_centers[i,:] = np.mean(X[y_db==i,:], axis = 0)

# plot the clusters and the centroids
plot_clusters(X,y_db,dim=(int_cols[0],int_cols[1]), points = cluster_centers)
```



2.3.1 Find the best parameters using ParameterGrid

[11]:
```
# prepare the parameters grid for `eps` and `min_samples`
# the ranges will include the default values
param_grid = {'eps': list(np.arange(0.1, 2, 0.1)), # 20 values
              'min_samples': list(range(3,10,1)) # 7 values
             }
params = list(ParameterGrid(param_grid))

# In order to avoid output cluttering, we will set thresholds for
# the silhouette and the percentage of clustered data
# Combinations under the threshold will not be printed
sil_thr = 0.7  # visualize results only for combinations
```

```
                # with silhouette above the threshold
clust_thr = 0.5 # visualize results only for combinations
                 # with ratio clustered/data above the threshold
```

[12]:
```
def inertia_and_centers(X, y):
    ''' Computes the centroids and the inertia given a set of points X and␣
↪labels y
        Parameters:
        X    : an array of shape[0] points in shape[1] dimensions
        y    : a vector of labels corresponding to the points, proper labels
              are in range 0..(shape[0]-1); possibly noise points are labelled␣
↪-1
        Uses the 'Einstein summation convention' from numpy for fast␣
↪computation of
        the inertia, see the [documentation](https://docs.scipy.org/doc/numpy/
↪reference/generated/numpy.einsum.html#numpy.einsum)
    '''
    cluster_labels_all = np.unique(y)
    cluster_labels = cluster_labels_all[cluster_labels_all != -1]
    n_clusters = len(cluster_labels)
    cluster_centers = np.empty((n_clusters,X.shape[1]))
    inertia = 0
    for i in cluster_labels:
        X_i = X[y==i,:]
        cluster_centers[i,:] = np.mean(X_i, axis = 0)
        X_i -= cluster_centers[i,:]
        inertia += np.einsum('ij,ij',X_i,X_i).mean()
    return inertia, cluster_centers, n_clusters
```

[13]:
```
# Prepare a row to describe formatted output
print("{:>11}\t{:>11}\t{:>11}\t{:>11}\t{:>11}\t{:>11}\t{:>11}".\
      format('eps','min_samp','n_clust','silh', 'clust_f', 'sc_index',␣
↪'inertia'))
# Prepare a dataframe for the results
results = pd.DataFrame(columns=['eps','min_samp',
                                'n_clust','silh', 'clust_f', 'sc_index',␣
↪'inertia'])
# for each combination of parameter values executes clustering
# then, saves the output if the quality measures are above the thresholds
for i in range(len(params)):
    # initialise, then fit_predict the estimator, obtaining
    # the labelling; unclustered points (noise) are labelled -1
    db = DBSCAN(**(params[i]))
    y_db = db.fit_predict(X)
    # list of the proper cluster labels (excluding noise)
#     cluster_labels_all = np.unique(y_db)
```

```
#     cluster_labels = cluster_labels_all[cluster_labels_all != -1]
    inertia, cluster_centers, n_clusters = inertia_and_centers(X,y_db)
#     n_clusters = len(cluster_labels)
    # filters out the unclustered points
    X_cl = X[y_db!=-1,:]
    y_db_cl = y_db[y_db!=-1]
    if n_clusters > 1 and n_clusters < X.shape[0]:
        # silhouette index cannot be computed out of this range
        silhouette = silhouette_score(X_cl,y_db_cl)
    else:
        # 0 or N clusters
        silhouette = -2
    clust = y_db_cl.shape[0]/y_db.shape[0] # fraction of clustered
    # sc_index = an index computed as the armonic mean between
    # Silhouette and the fraction of clustered
    # (it is just a suggestion, it is not a standard measure, and
    #  it was not requested for the exercise)
    sc_index = silhouette*clust/(silhouette+clust)
    if silhouette > sil_thr and clust > clust_thr:
        # if above threshold save the results
        results = results.append({'eps':db.eps,
                              'min_samp':db.min_samples,
                              'n_clust':n_clusters,
                              'silh':silhouette,
                              'clust_f':clust,
                              'sc_index':sc_index,
                              'inertia':inertia,
                             },
                             ignore_index = True)
        print("{:11.2f}\t{:11}\t{:11}\t{:11.2f}\t{:11.2f}\t{:11.2f}\t{:11.2f}"\
              .format(db.eps, db.min_samples, n_clusters,
                      silhouette, clust, sc_index, inertia))
```

```
        eps        min_samp        n_clust            silh          clust_f
sc_index          inertia
       0.40               6              7            0.73             0.56
0.32           465.39
       0.40               7              5            0.77             0.53
0.31           415.35
       0.40               8              6            0.79             0.50
0.31           379.08
       0.50               8              7            0.71             0.66
0.34           666.54
       0.50               9              6            0.71             0.63
0.33           615.01
       0.60               7              9            0.73             0.82
0.38          1151.12
```

0.60	8	7	0.76	0.79	0.39	1064.85
0.60	9	7	0.78	0.76	0.39	1000.91
0.70	7	6	0.70	0.91	0.40	1966.82
0.70	8	6	0.71	0.90	0.40	1892.89
0.80	3	5	0.71	0.97	0.41	3004.98
0.80	4	4	0.76	0.96	0.42	2968.79
0.80	5	4	0.76	0.96	0.42	2954.30
0.80	8	4	0.77	0.94	0.42	2641.71
0.80	9	4	0.77	0.94	0.42	2508.72
0.90	3	4	0.75	0.97	0.42	3214.11
0.90	4	4	0.76	0.97	0.42	3143.04
0.90	5	4	0.76	0.97	0.42	3058.92
0.90	6	4	0.76	0.97	0.43	3034.91
0.90	7	4	0.76	0.96	0.42	3003.08
0.90	8	4	0.76	0.96	0.43	2971.91
0.90	9	4	0.76	0.96	0.43	2921.37
1.00	5	4	0.75	0.97	0.42	3189.40
1.00	6	4	0.76	0.97	0.42	3113.40
1.00	7	4	0.76	0.97	0.42	3113.40
1.00	8	4	0.76	0.97	0.42	3113.40
1.00	9	4	0.76	0.97	0.43	3063.74
1.10	4	4	0.75	0.99	0.42	3517.21
1.10	5	4	0.75	0.98	0.42	3459.19
1.10	6	4	0.75	0.98	0.43	3318.16

```
        1.10                7                4            0.75            0.98
0.43          3268.05
        1.10                8                4            0.75            0.98
0.43          3241.27
        1.10                9                4            0.75            0.98
0.43          3241.27
        1.20                7                4            0.75            0.98
0.42          3450.50
        1.20                8                4            0.75            0.98
0.42          3423.50
        1.20                9                4            0.75            0.98
0.43          3353.96
        1.30                9                4            0.75            0.99
0.42          3500.42
```

2.4 4. Plot of the inertia and silhouette

With DBSCAN the number of clusters is a dependent variable, being the independent ones `eps` and `min_samples`. For this plot we will find, for each number of clusters, the combination of independent variables guaranteing the best silhouette index, using the thresholds filtered combinations stored in `results`.

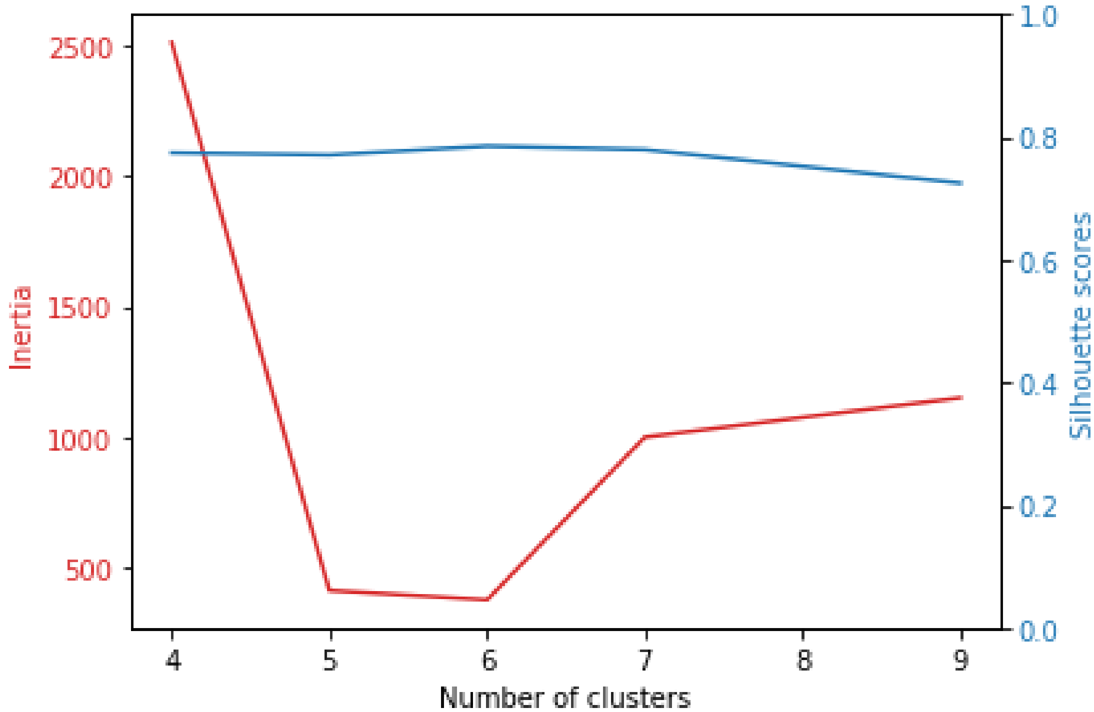
The plot will not consider the noise points, otherwise the results would be obviously misleading.

[14]:
```
# for each resulting number of clusters find the results row
#    for which the silhouette index is maximum
idx_max_silh = results.groupby(['n_clust'])['silh'].transform(max) ==␣
↪results['silh']
```

[15]:
```
top_results = results[idx_max_silh].sort_values('n_clust')
```

[16]:
```
fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('Number of clusters')
ax1.set_ylabel('Inertia', color=color)
ax1.plot(top_results['n_clust'].values, top_results['inertia'].values,␣
↪color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('Silhouette scores', color=color)  # we already handled the␣
↪x-label with ax1
ax2.plot(top_results['n_clust'].values, top_results['silh'].values, color=color)
ax2.tick_params(axis='y', labelcolor=color)
```

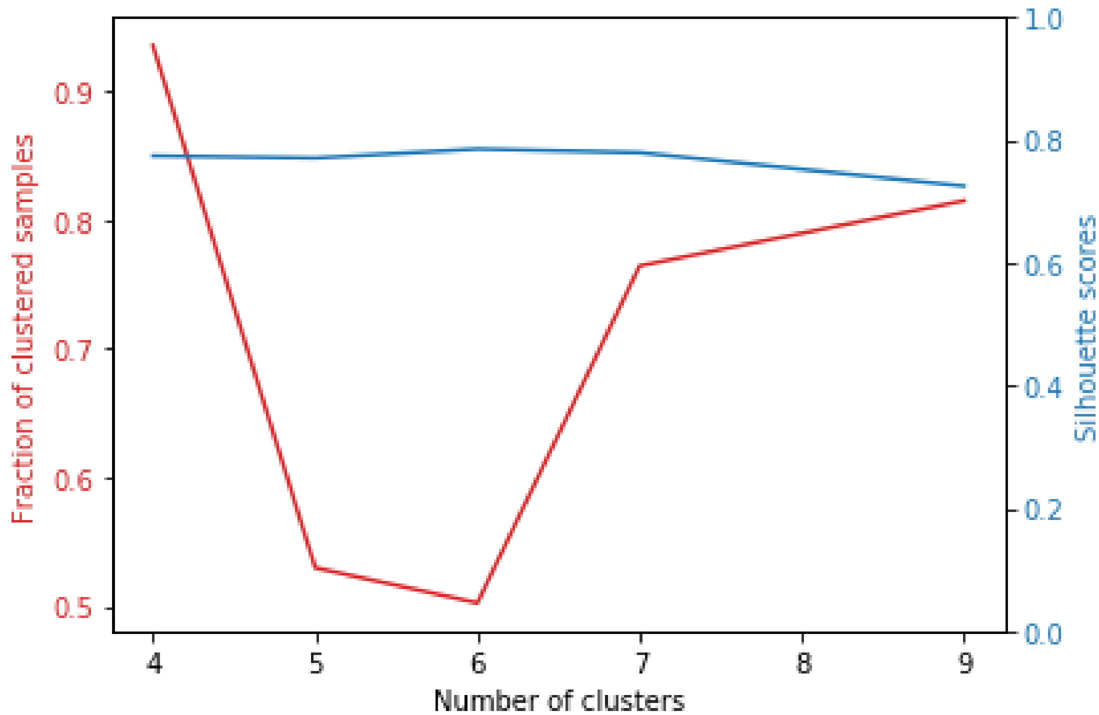
```
ax2.set_ylim(0,1) # the axis for silhouette is [0,1]

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
```



For DBSCAN the *inertia* can be misleading, since it is naturally decreased for increasing number of *noise* samples. The *fraction of clustered samples* is probably more interesting in this case.

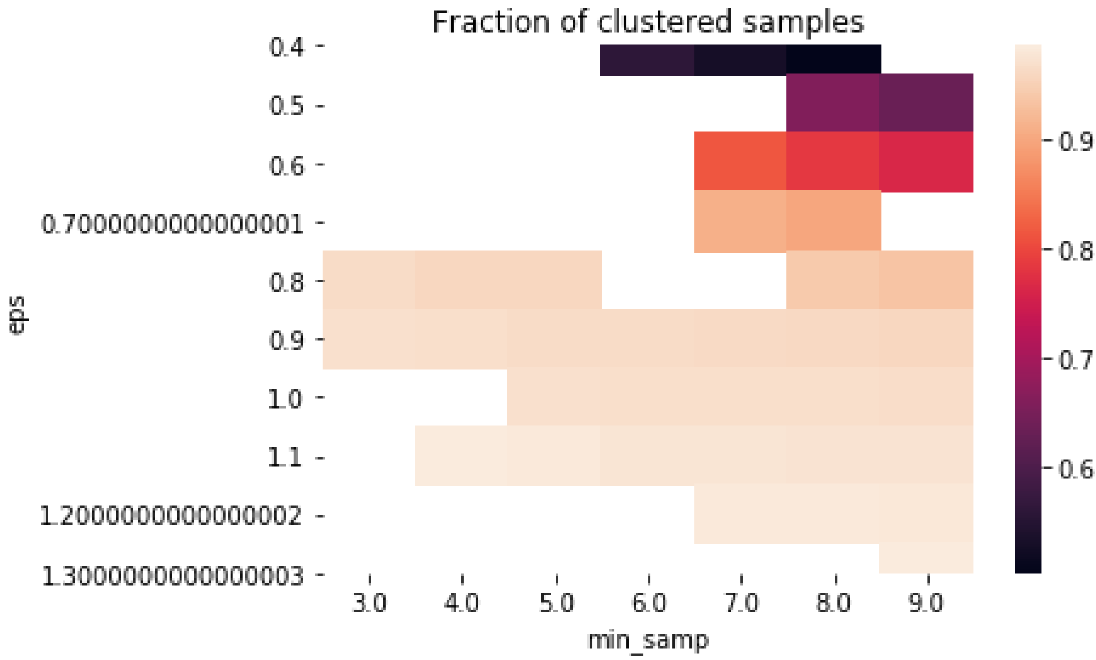
[17]:
```
fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('Number of clusters')
ax1.set_ylabel('Fraction of clustered samples', color=color)
ax1.plot(top_results['n_clust'].values, top_results['clust_f'].values,␣
 ↪color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('Silhouette scores', color=color)  # we already handled the␣
 ↪x-label with ax1
ax2.plot(top_results['n_clust'].values, top_results['silh'].values, color=color)
```

```
ax2.tick_params(axis='y', labelcolor=color)
ax2.set_ylim(0,1) # the axis for silhouette is [0,1]

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
```

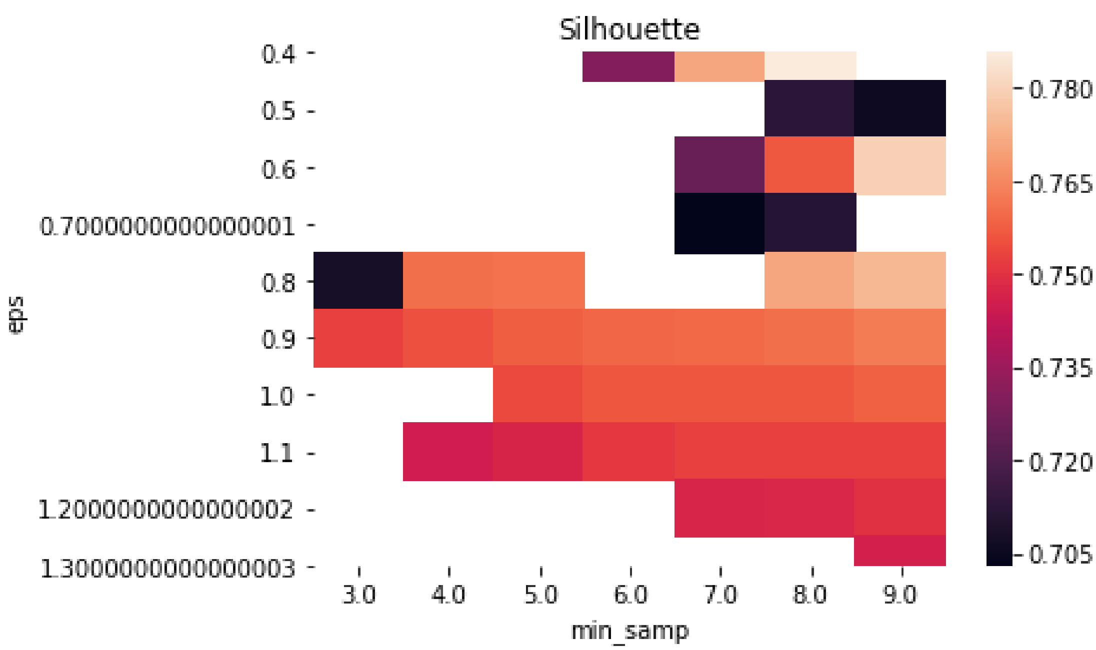


Additional insights can be obtained by the `heatmaps`, which allow to observe the results of pairs of parameters.

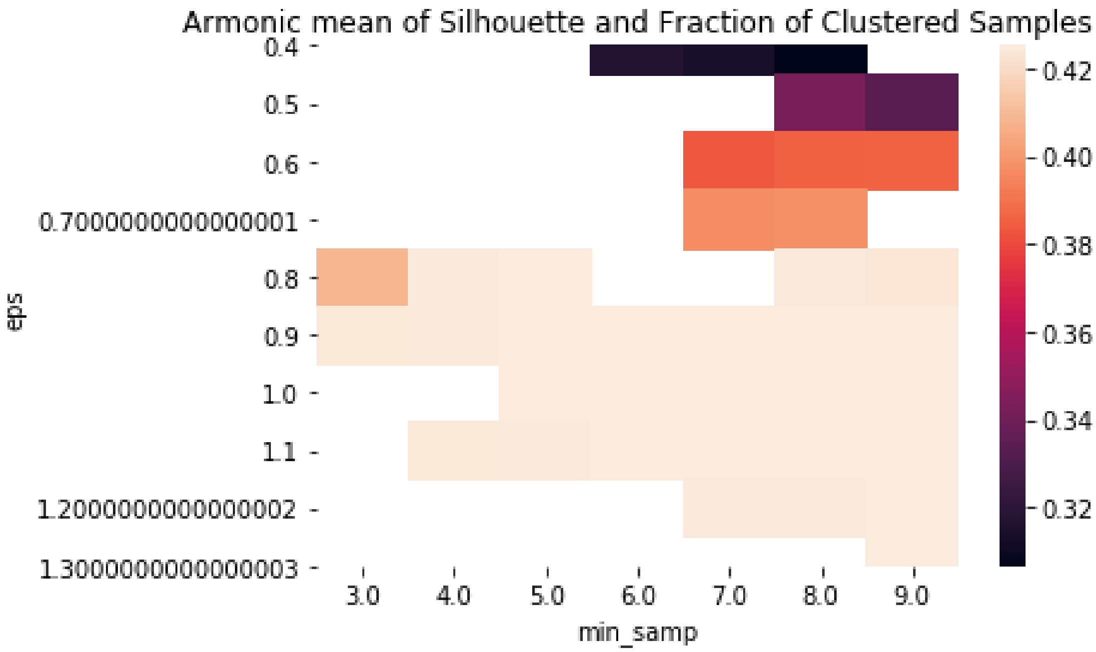
[18]:
```
results_pivot =results.pivot('eps', 'min_samp', 'clust_f')
plt.title("Fraction of clustered samples")
ax = sns.heatmap(results_pivot)
```



[19]:
```
import seaborn as sns
results_pivot =results.pivot('eps', 'min_samp', 'silh')
plt.title("Silhouette")
ax = sns.heatmap(results_pivot)
```



[20]:
```
results_pivot =results.pivot('eps', 'min_samp', 'sc_index')
plt.title("Armonic mean of Silhouette and Fraction of Clustered Samples")
ax = sns.heatmap(results_pivot)
```



2.5 5. Show the best value for the parameter(s)

[21]:
```
# find the row with the best results according to
# sc_index
best = results.iloc[results.index.argmax()]
```

[22]:
```
print("Best tradeoff between Silhouette and Fraction of Clustered")
print("EPS = {:3.2f}\tMIN_SAMPLES = {:2.0f}".format(best.eps, best.min_samp))
print("Results with best parameters")
print("Silhouette = {:3.2f}\tClustered Fraction = {:3.2f}\tNumber of Clusters =␣
↪{:2.0f}".\
      format(best.silh, best.clust_f,best.n_clust))
```

```
Best tradeoff between Silhouette and Fraction of Clustered
EPS = 1.30      MIN_SAMPLES =  9
Results with best parameters
Silhouette = 0.75       Clustered Fraction = 0.99       Number of Clusters =  4
```

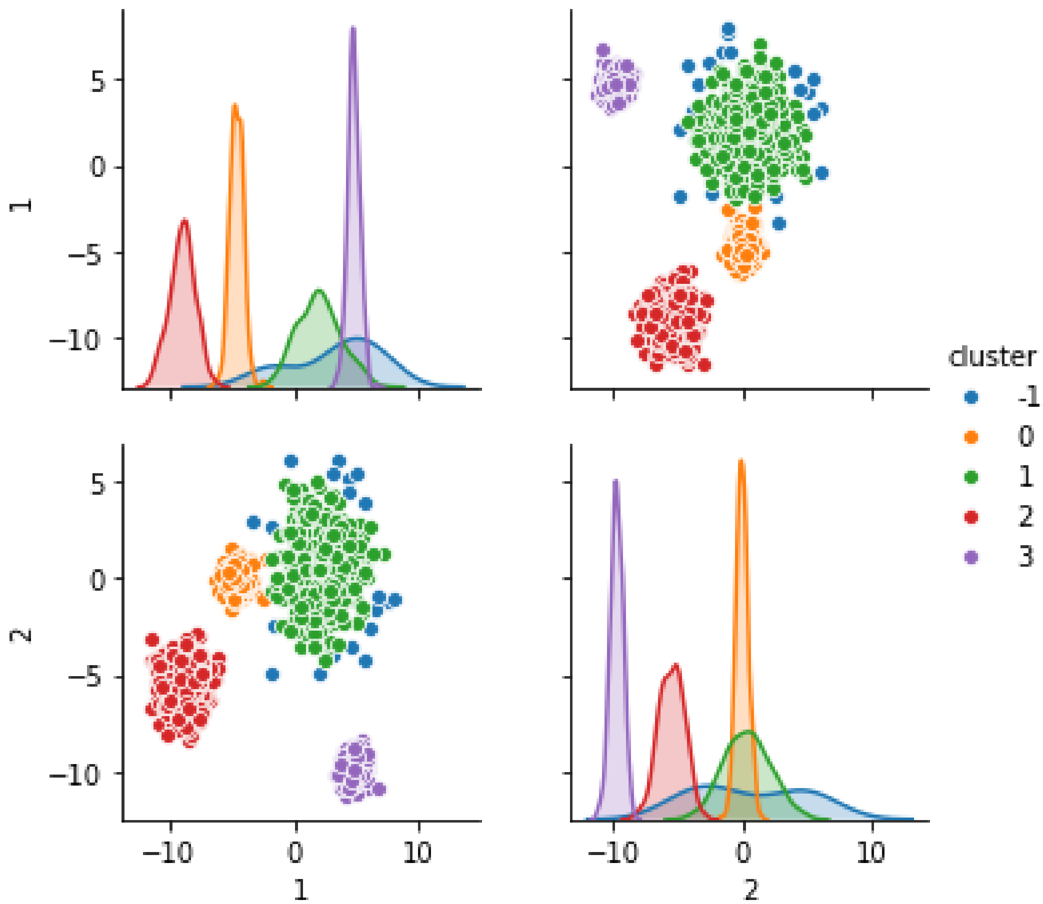
2.6 6. Cluster with the optimal parameter(s) and show the pairplot

[23]:
```
db = DBSCAN(eps=best.eps, min_samples=int(best.min_samp))
y_db = db.fit_predict(X)
```

[24]:
```
df['cluster'] = y_db
```

[25]:
```
sns.pairplot(df[int_cols + ['cluster']], vars = df[int_cols], hue = 'cluster')
```

[25]: <seaborn.axisgrid.PairGrid at 0x1a1d986850>



[26]:
```
# plot_clusters(X,y_db,dim=(int_cols[0],int_cols[1]), points = cluster_centers)
```

2.7 7. Quantifying the quality of clustering via silhouette plots

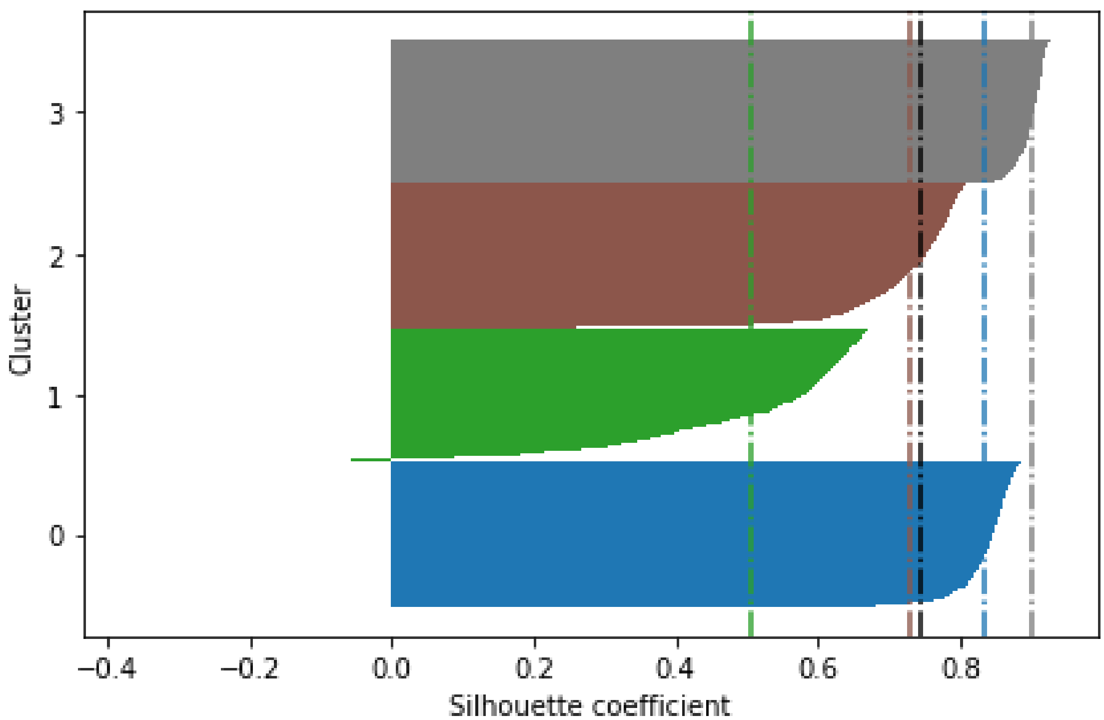
The silhouette scores for the individual samples are computed with the function `silhouette_samples`

The function `plot_silhouette` produces a 'horizontal bar-plot', with one bar for each sample, where the length of the bar is proportional to the silhouette score of the sample. The bars are grouped for cluster and sorted for decreasing length.

A vertical line represents the silhouette score, i.e. the average on all the samples.

[27]:
```
# from plot_silhouette import plot_silhouette
from plot_silhouette2 import plot_silhouette
```

[28]:
```
X_cl = X[y_db!=-1,:]
y_db_cl = y_db[y_db!=-1]
silhouette = silhouette_samples(X_cl,y_db_cl)
plot_silhouette(silhouette,y_db_cl)
# silhouette = silhouette_samples(X,y_db)
# plot_silhouette(silhouette,y_db)
```



2.8 8. Sorted list of the discovered clusters for decreasing sizes (7pt)

`groupby().size()` returns a dataframe, resetting the index we can name the newly created column with the counts

[29]:
```
counts = pd.DataFrame(y_db).groupby(0).size().reset_index(name='Count')
print(counts)
```

```
   0  Count
0 -1     20
1  0    378
2  1    352
3  2    375
4  3    375
```

[30]:
```
counts = pd.DataFrame(y_db).groupby(0).size().reset_index(name='Count')
```

Then we rename the first column as `Cluster`, sort by descending values of the counts and print the dataframe without showing the index

[31]:
```
counts = counts.rename(columns = {0: 'Cluster'}).sort_values(by = 'Count',␣
 ↪ascending = False)
# print(counts[counts.Cluster!=-1].to_string(index = False))
print(counts.to_string(index = False))
```

```
 Cluster  Count
       0    378
       2    375
       3    375
       1    352
      -1     20
```

ml-lab17-12-2019-KMeans

February 12, 2020

Claudio Sartori©

1 Machine Learning - Lab

1.1 Example of Lab exam

1.1.1 Find the best number of clusters with k__means

2 Tasks

Find the clusters in the included dataset.

The solution must be produced as a Python Notebook. The notebook must include appropriate comments and must produce:

1. the boxplots of the attributes and a comment on remarkable situations, if any (2pt)
2. a pairplot of the data (see Seaborn pairplot) and a comment on remarkable situations, if any (2pt)
3. a clustering schema using a method of your choice exploring a range of parameter values (5pt)
4. the plot of the global inertia (SSD) and silhouette index for the parameter values you examine (4pt)
5. the optimal parameters of your choice (4pt)
6. a pairplot of the data using as hue the cluster assignment with the optimal parameter (3pt)
7. a plot of the silhouette index for the data points, grouped according to the clusters (4pt)
8. A sorted list of the discovered clusters for decreasing sizes (7pt)

[1]:
```
from IPython.display import Image
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, silhouette_samples

%matplotlib inline
```

```
rnd_state = 42 # This variable will be used in all the procedure calls allowing␣
↪a random_state parameter
               # in this way the running can be perfectly reproduced
               # just change this value for a different experiment
```

[2]:
```
data_file = 'lab_exercise.csv'
delimiter = ','
X = np.loadtxt(data_file, delimiter = delimiter)
```

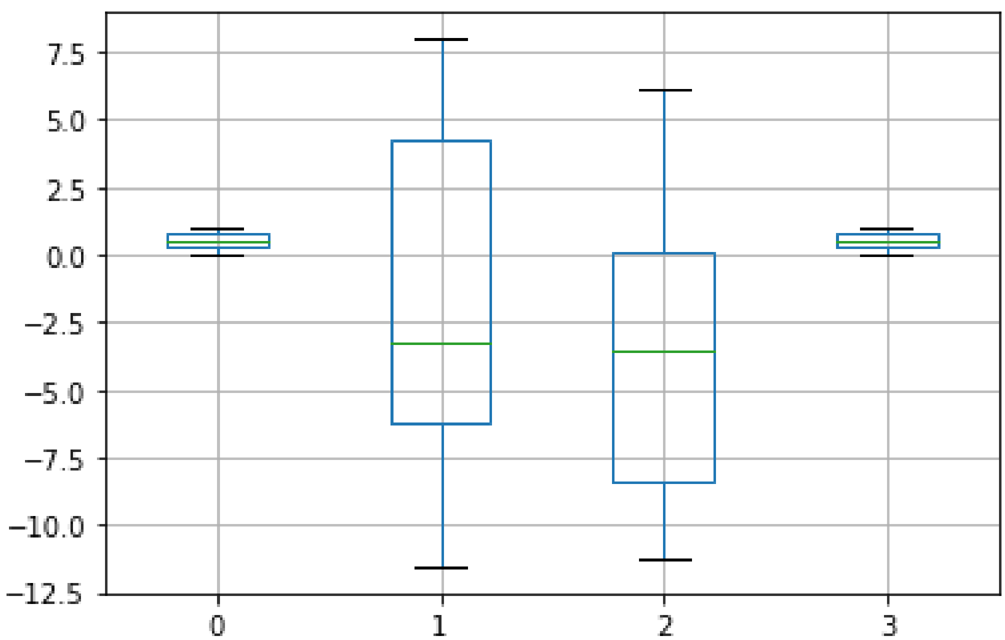
[3]:
```
X.shape
```

[3]: (1500, 4)

In order to exploit the Pandas DataFrame features we generate `df` from `X`

[4]:
```
df = pd.DataFrame(X)
df.boxplot()
```

[4]: <matplotlib.axes._subplots.AxesSubplot at 0x10ef8de90>



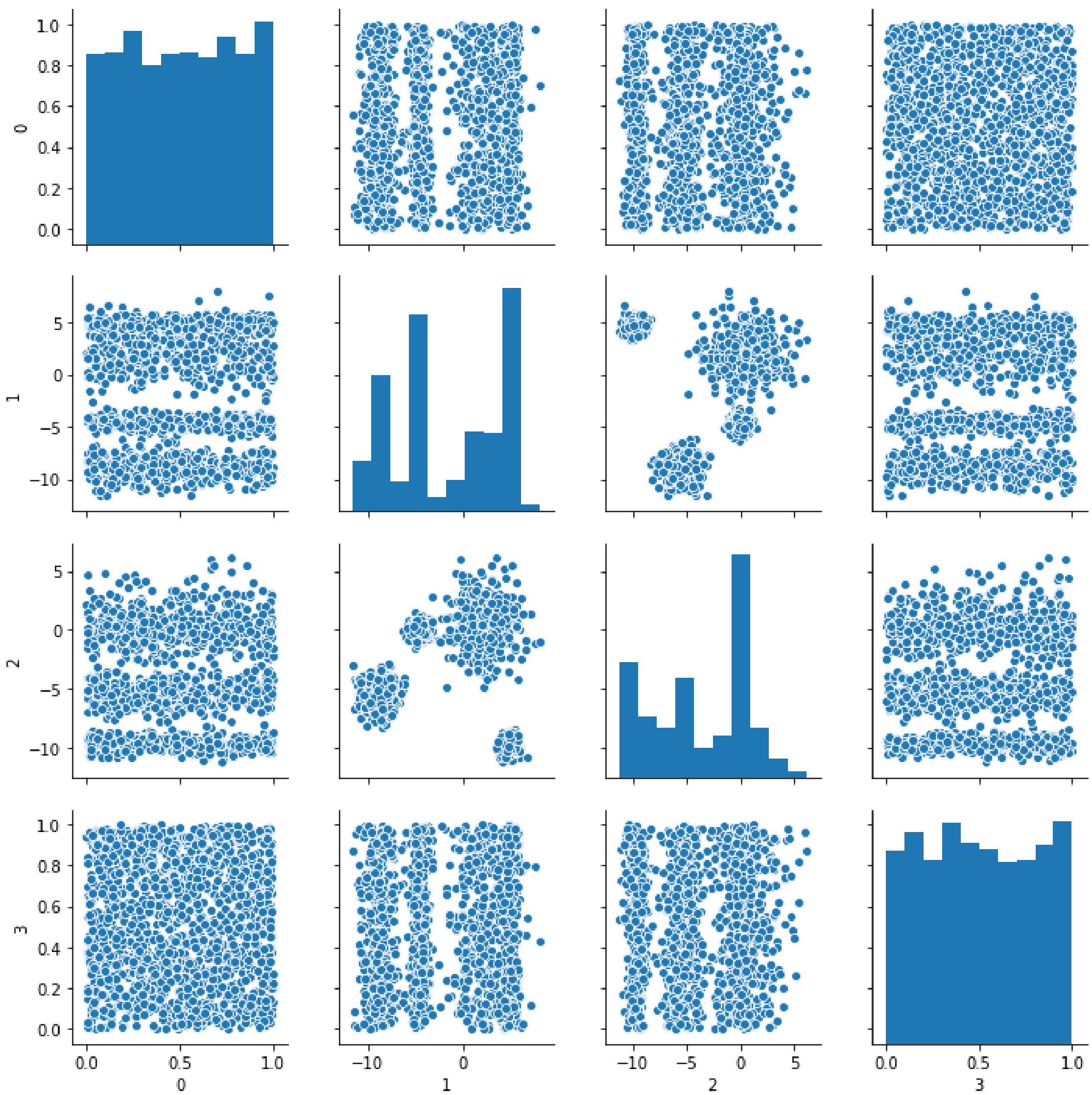
2.1 1. Comment on boxplots

Columns 0 and 3 have a range much smaller than 1 and 2. The distributions onf 0 and 3 seem to be equal. Poddibly, a min-max rescaling could point out some additional insight.

Let's look at the pairplots and consider if it is worth to do this transformation.

[5]:
```
sns.pairplot(df)
```

[5]: <seaborn.axisgrid.PairGrid at 0x1a236ded50>



2.2 2. Comments on pairplots

The pairplots show that the two most interesting columns are `1` and `2`, their pairplot shows evident clusters.

The pairplots of `0` and `3` show that those columns are uniformly distributed and do not show any pattern.

[6]:
```
int_cols = [1,2] # Interesting columns
```

2.3 3. Find a clustering scheme with KMeans

We will try **k_means** with a number of clusters varying from 2 to 10

- prepare two emptys lists for inertia and silhouette scores
- For each value of the number of clusters:

- initialize an estimator for `KMeans` and `fit_predict`
- we will store the distortion (from the fitted model) in the variable `distortions`
- using the function `silhouette_score` from `sklearn.metrics` with arguments the data and the fitted labels, we will fill the variable `silhouette_scores`

Then we will plot the two lists in the y axis, with the range of k in the x axis. The plot with two different scales in the `y` axis can be done according to the example shown in the notebook `two_scales.ipynb`.

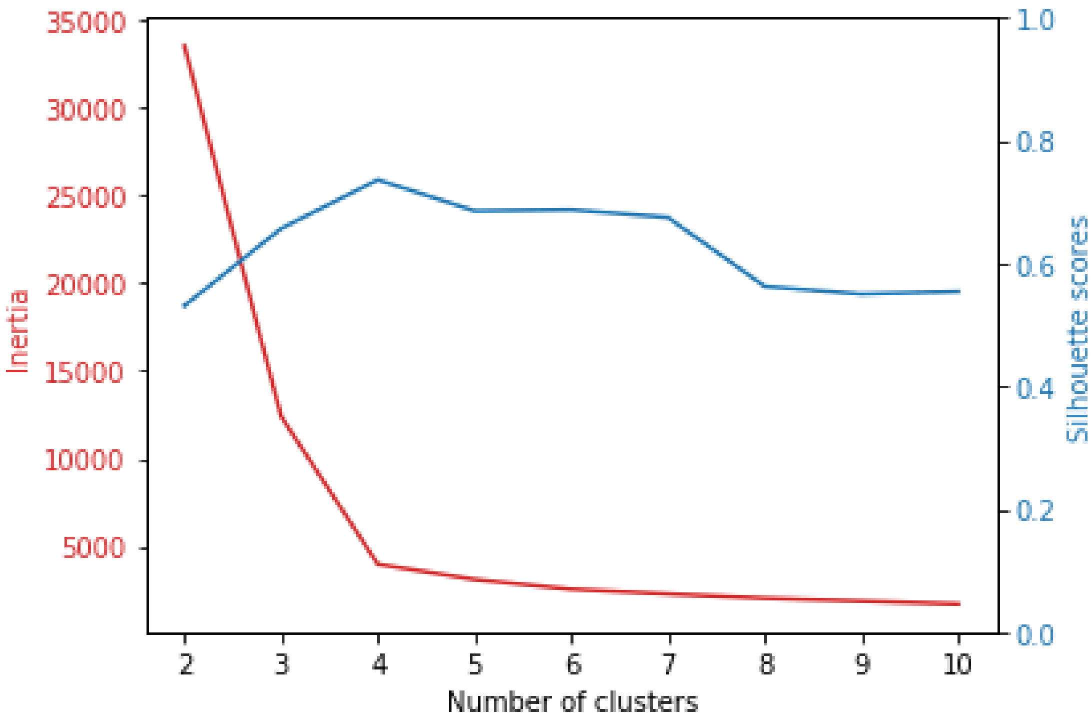
[7]:
```
k_range = range(2,11)
```

[8]:
```
distortions = []
silhouette_scores = []
for i in k_range:
    km = KMeans(n_clusters=i,
                init='k-means++',
                n_init=10,
                max_iter=300,
                random_state=rnd_state)
    y_km = km.fit_predict(X)
    distortions.append(km.inertia_)
    silhouette_scores.append(silhouette_score(X,y_km))
```

2.4 4. Plot of inertia and silhouette

[9]:
```
fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('Number of clusters')
ax1.set_ylabel('Inertia', color=color)
ax1.plot(k_range, distortions, color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
```

```
ax2.set_ylabel('Silhouette scores', color=color)  # we already handled the␣
↪x-label with ax1
ax2.plot(k_range, silhouette_scores, color=color)
ax2.tick_params(axis='y', labelcolor=color)
ax2.set_ylim(0,1) # the axis for silhouette is [0,1]

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
```



2.5 5. Show the best value for the parameter(s)

[10]:
```
good_k = np.argmax(silhouette_scores) + k_range.start
print("The value of K providing the maximum silhouette index is {}".
↪format(good_k))
```

```
The value of K providing the maximum silhouette index is 4
```

2.6 6. Cluster with the optimal parameter(s) and show the pairplot

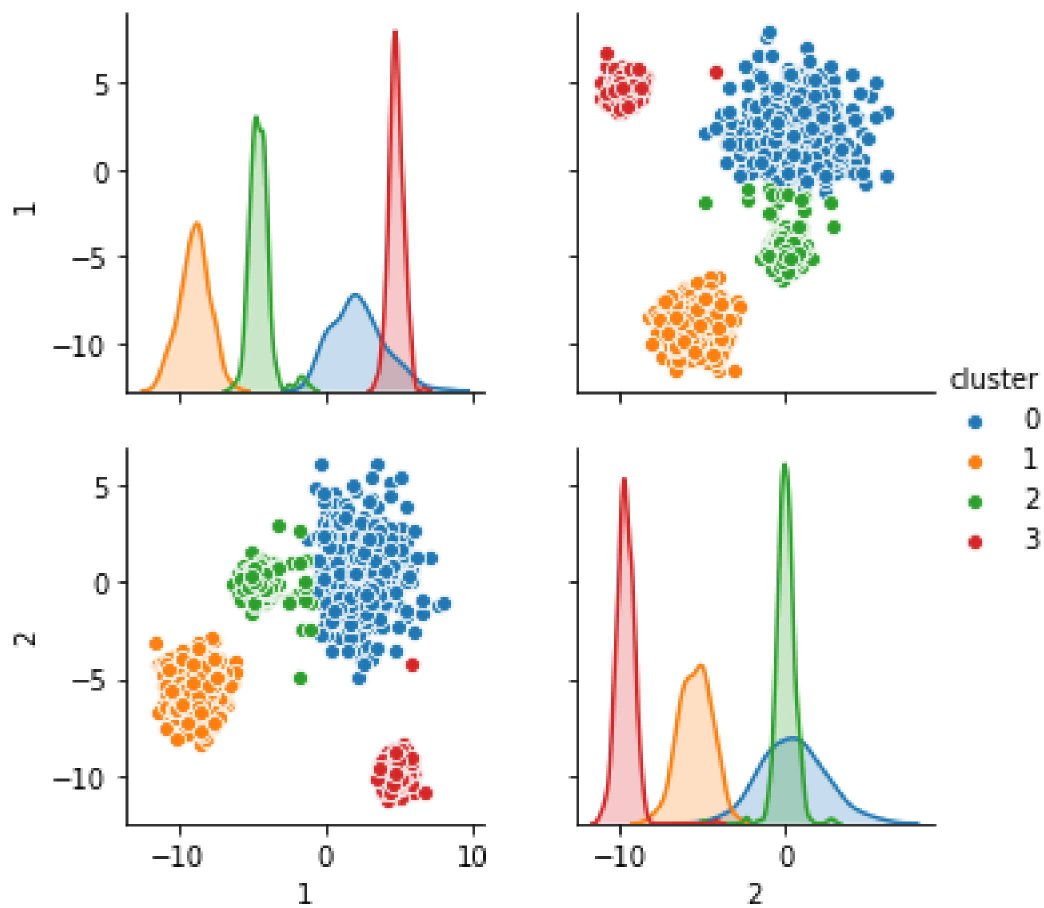
[11]:
```
km = KMeans(n_clusters=good_k,
            init='k-means++',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=rnd_state)
y_km = km.fit_predict(X)
```

In order to show the ”hue” with the Seaborn pairplot it is necessary to add the `cluster` column and then specify as `vars` the data columns you want to plot.

[12]:
```
df['cluster'] = y_km
```

[13]:
```
sns.pairplot(df[int_cols + ['cluster']], vars = df[int_cols], hue = 'cluster')
```

[13]:
```
<seaborn.axisgrid.PairGrid at 0x1a240ae890>
```



2.7 7. Quantifying the quality of clustering via silhouette plots

The silhouette scores for the individual samples are computed with the function `silhouette_samples`

The function `plot_silhouette` produces a 'horizontal bar-plot', with one bar for each sample, where the length of the bar is proportional to the silhouette score of the sample. The bars are grouped for cluster and sorted for decreasing length.

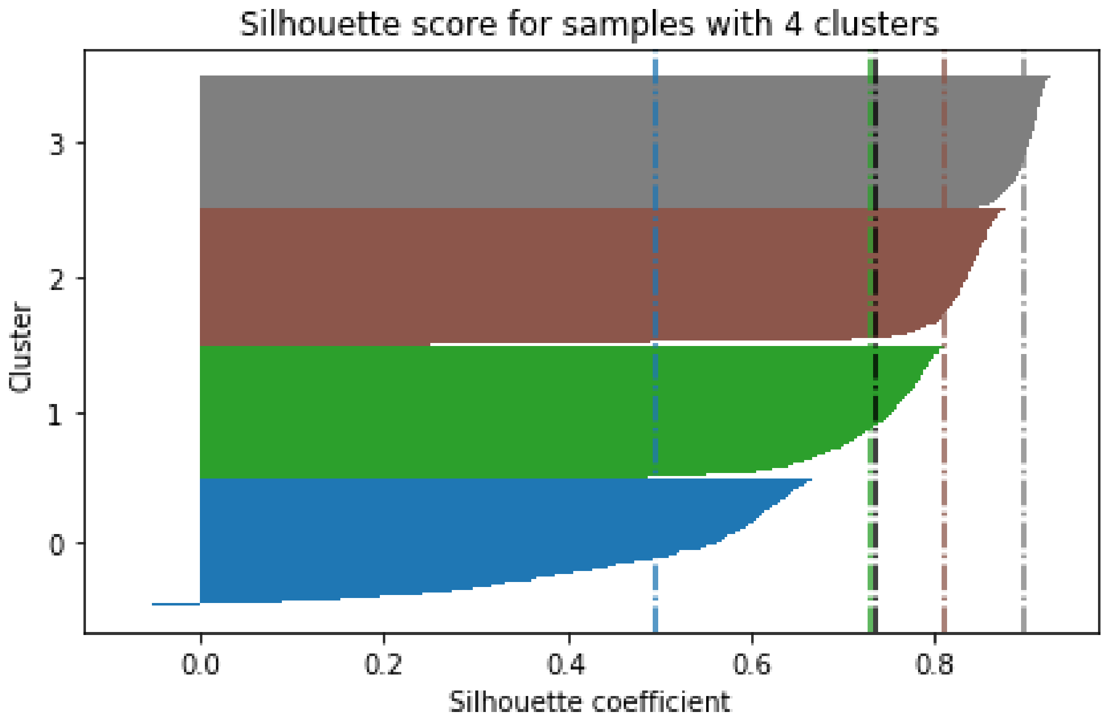
A vertical line represents the silhouette score, i.e. the average on all the samples

[14]:
```
# from plot_silhouette import plot_silhouette
from plot_silhouette2 import plot_silhouette
```

[15]:
```
help(plot_silhouette)
```

```
Help on function plot_silhouette in module plot_silhouette2:

plot_silhouette(silhouette_vals, y, colors=<matplotlib.colors.ListedColormap
object at 0x11d48a0d0>, plot_noise=False)
    Plotting silhouette scores for the individual samples of a labelled data
set.
    The scores will be grouped according to labels and sorted in descending
order.
    The bars are proportional to the score and the color is determined by the
label.

    silhouette_vals: the silhouette values of the samples
    y:               the labels of the samples
    plot_noise:      boolean, assumes the noise to be labeled with a negative
integer
```

[16]:
```
cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0] # it is the number of rows
# Compute the Silhouette Coefficient for each sample, with the euclidean metric
silhouette_score_samples = silhouette_samples(X, y_km, metric='euclidean')
plt.title('Silhouette score for samples with {} clusters'.format(good_k))
plot_silhouette(silhouette_score_samples, y_km)
```



2.8 8. Sorted list of the discovered clusters for decreasing sizes (7pt)

`groupby().size()` returns a dataframe, resetting the index we can name the newly created column with the counts

[17]:
```
counts = pd.DataFrame(y_km).groupby(0).size().reset_index(name='Count')
print(counts)
```

```
   0  Count
0  0    359
1  1    375
2  2    390
3  3    376
```

Then we rename the first column as `Cluster`, sort by descending values of the counts and print the dataframe without showing the index

[18]:
```
counts = counts.rename(columns = {0: 'Cluster'}).sort_values(by = 'Count',␣
 ↪ascending = False)
print(counts.to_string(index = False))
```

```
 Cluster  Count
       2    390
       3    376
```

1	375
0	359