Data_python_professor

September 19, 2019

1 Lab of data analysis with python

```
Author: Jesús Fernández Bes
        Jerónimo Arenas García (jeronimo.arenas@uc3m.es)
        Jesús Cid Sueiro (jcid@tsc.uc3m.es)

Notebook version: 1.1 (Sep 20, 2017)

Changes: v.1.0 - First version.
         v.1.1 - Compatibility with python 2 and python 3

Pending changes:
```

In this lab we will introduce some of the modules that we will use in the rest of the labs of the course.

The usual beginning of any python module is a list of import statements. In most our file we will use the following modules:

- numpy: The basic scientific computing library.
- csv: Used for input/output in using comma separated values files, one of the standards formats in data management.
- matplotlib: Used for plotting figures and graphs
- sklearn: Scikit-learn is the machine learning library for python.

[1]:
```
%matplotlib inline
# Needed to include the figures in this notebook, you can remove it
# to work with a normal script

import numpy as np
import csv
import matplotlib.pyplot as plt

from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
```

```
/Users/jcid/anaconda/envs/mypy36/lib/python3.6/site-
packages/sklearn/cross_validation.py:44: DeprecationWarning: This module was
deprecated in version 0.18 in favor of the model_selection module into which all
the refactored classes and functions are moved. Also note that the interface of
the new CV iterators are different from that of this module. This module will be
removed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

1.1 1. NUMPY

The *numpy* module is useful for scientific computing in Python.

The main data structure in *numpy* is the n-dimensional array. You can define a *numpy array* from a list or a list of lists. Python will try to build it with the appropiate dimensions. You can check the dimensions of the array with *shape()*

[2]:
```
my_array = np.array([[1, 2],[3, 4]])
print(my_array)
print(np.shape(my_array))
```

```
[[1 2]
 [3 4]]
(2, 2)
```

Define a new 3x2 array named *my_array2* with [1, 2, 3] in the first row and [4,5,6] in the second. Check the dimension of the array.

[3]:
```
#<SOL>
my_array2 = np.array([[1,2,3],[4,5,6]])
print(my_array2)
print(np.shape(my_array2))
#</SOL>
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
```

There are a number of operations you can do with numpy arrays similar to the ones you can do with matrices in Matlab. One os the most important is **slicing**. We saw it when we talked about lists, it consists in extracting some subarray of the array.

[4]:
```
my_array3 = my_array[:,1]
print(my_array3)
print(my_array[1,0:2])
```

```
[2 4]
[3 4]
```

One important thing to consider when you do slicing are the dimensions of the output array. Check the shape of *my_array3*. Check also its dimension with function *ndim*:

[5]:
```
#<SOL>
print(np.shape(my_array3))
print(np.ndim(my_array3))
#</SOL>
```

```
(2,)
1
```

If you have correctly computed it you will see that *my_array3* is one dimensional. Sometimes this can be a problem when you are working with 2D matrixes (and vectors can be considered as 2D matrixes with one of the sizes equal to 1). To solve this, *numpy* provides the *newaxis* constant.

[6]:
```
my_array3 = my_array3[:,np.newaxis]
```

Check again the shape and dimension of *my_array3*

[7]:
```
#<SOL>
print(my_array3)
print(np.shape(my_array3))
print(np.ndim(my_array3))
#</SOL>
```

```
[[2]
 [4]]
(2, 1)
2
```

It is possible to extract a single row or column from a 2D numpy array so that the result is still 2D, without explictly recurring to *np.newaxis*. Compare the outputs of the following print commands.

[8]:
```
print(my_array[:,1])
print(my_array[:,1].shape)
print(my_array[:,1:2])
print(my_array[:,1:2].shape)
```

```
[2 4]
(2,)
[[2]
 [4]]
(2, 1)
```

Another important array manipulation method is array *concatenation* or *stacking*. It is useful to always state explicitly in which direction we want to stack the arrays. For example in the following example we are stacking the arrays vertically.

[9]:
```
print(my_array)
print(my_array2)
print(np.concatenate( (my_array, my_array2) , axis=1)) # columnwise concatenation
```

```
[[1 2]
 [3 4]]
[[1 2 3]
 [4 5 6]]
[[1 2 1 2 3]
 [3 4 4 5 6]]
```

EXERCISE: Concatenate the first column of *my_array* and the second column of *my_array2*

[10]:
```
#<SOL>
print(np.concatenate( (my_array[:,0:1], my_array2[:,1:2]) , axis=1))
#</SOL>
```

```
[[1 2]
 [3 5]]
```

You can create *numpy* arrays in several ways, not only from lists. For example *numpy* provides a number of functions to create special types of matrices.

EXERCISE: Create 3 arrays usings *ones*, *zeros* and *eye*. If you have any doubt about the parameters of the functions have a look at the help with the function *help( )*.

[11]:
```
#<SOL>
ones_array = np.ones((3,2))
zeros_array = np.zeros((3,2))
eye_array = np.eye(3)
print(ones_array)
print(zeros_array)
print(eye_array)
#</SOL>
```

```
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
[[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Finally *numpy* provides all the basic matrix operations: multiplications, dot products, ... You can find information about them in the Numpy manual.

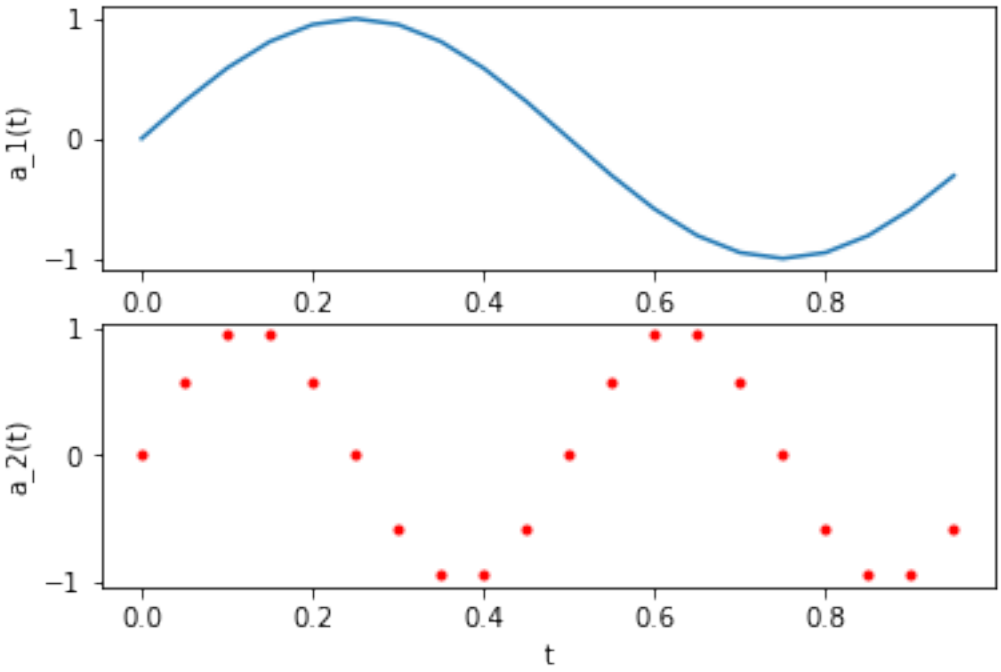
In addition to *numpy* we have a more advanced library for scientific computing, Scipy, that includes modules for linear algebra, signal processing, Fourier transform, ...

1.2 2. Matplotlib

One important step of data analysis is data visualization. In python the simplest plotting library is *matplotlib* and its sintax is similar to Matlab plotting library. In the next example we plot two sinusoids with different simbols.

[12]:
```
t = np.arange(0.0, 1.0, 0.05)
a1 = np.sin(2*np.pi*t)
a2 = np.sin(4*np.pi*t)
#s = sin(2*3.14159*t)

plt.figure()
ax1 = plt.subplot(211)
ax1.plot(t,a1)
plt.xlabel('t')
plt.ylabel('a_1(t)')
ax2 = plt.subplot(212)
ax2.plot(t,a2, 'r.')
plt.xlabel('t')
plt.ylabel('a_2(t)')
plt.show()
```



1.3 3. Classification example

One of the main machine learning problems is clasification. In the following example, we will load and visualize a dataset that can be used in a clasification problem.

The iris dataset is one of the most popular pattern recognition datasets. It consists on 150 instances of 4 features of iris flowers:

1. sepal length in cm

2. sepal width in cm
3. petal length in cm
4. petal width in cm

The objective is usually to distinguish three different classes of iris plant: Iris setosa, Iris versicolor, and Iris virginica.

1.3.1 3.1 Loading the data

We give you the data in *.csv* format. In each line of the csv file we have the 4 real-valued features of each instance and then a string defining the class of that instance: Iris-setosa, Iris-versicolor or Iris-virginica. There are 150 instances of flowers in the csv file.

Let's se how we can load the data in an *array*

[13]:
```
# Open up the csv file in to a Python object
csv_file_object = csv.reader(open('iris_data.csv', 'r'))
datalist = []                    # Create a variable called 'data'.
for row in csv_file_object:      # Run through each row in the csv file,

    datalist.append(row)         # adding each row to the data variable


data = np.array(datalist)  # Then convert from a list to an array
                           # Be aware that each item is currently
                           # a string in this format
print(np.shape(data))
X = data[:,0:-1]
label = data[:,-1,np.newaxis]
print(X.shape)
print(label.shape)
```

```
(150, 5)
(150, 4)
(150, 1)
```

In the previous code we have saved the features in matrix X and the class labels in the vector labels. Both are 2D *numpy arrays*. We are also printing the shapes of each variable (see that we can also use `array_name.shape` to get the shape, appart from function *shape()*). Checking the shape of matrices is a convenient way to prevent mistakes in your code.

1.3.2 3.2 Visualizing the data

Extract the 2 first features of the data (sepal length and width) and plot the first versus the second in a figure, use a different color for the data corresponding to different classes.

First of all you probably want to split the data according to each class label.

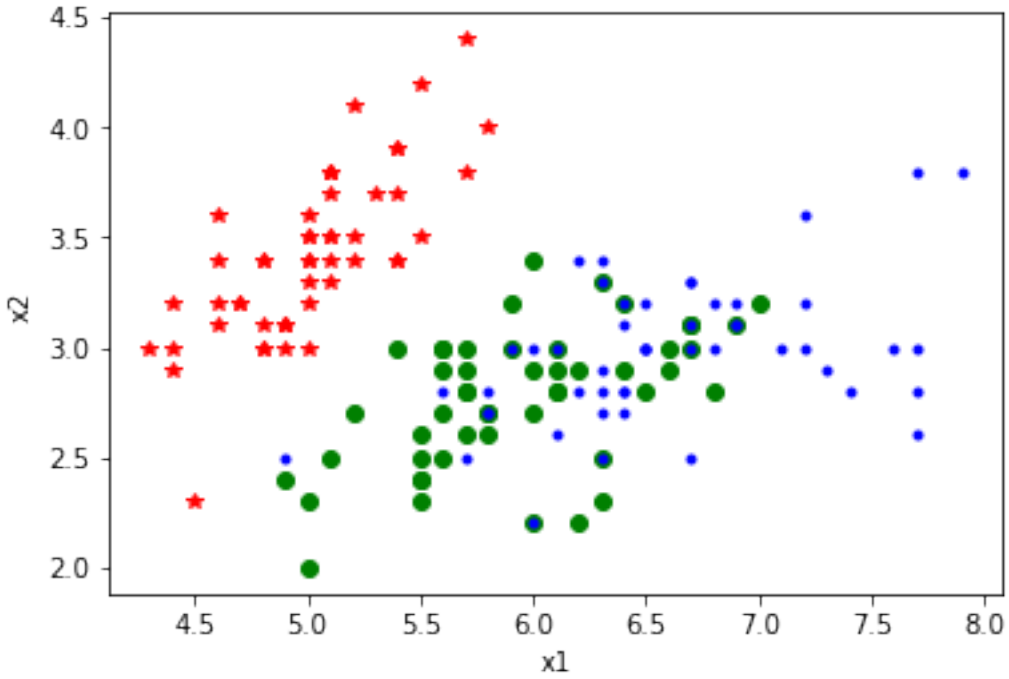
[14]:
```
#<SOL>
x1 = X[:,0]
x2 = X[:,1]
```

```
x1_1 = [x for i,x in enumerate(x1) if label[i]=='Iris-setosa' ]
x1_2 = [x for i,x in enumerate(x1) if label[i]=='Iris-versicolor' ]
x1_3 = [x for i,x in enumerate(x1) if label[i]=='Iris-virginica' ]
x2_1 = [x for i,x in enumerate(x2) if label[i]=='Iris-setosa' ]
x2_2 = [x for i,x in enumerate(x2) if label[i]=='Iris-versicolor' ]
x2_3 = [x for i,x in enumerate(x2) if label[i]=='Iris-virginica' ]

plt.figure()
plt.plot(x1_1,x2_1 , 'r*')
plt.hold(True)
plt.plot(x1_2,x2_2 , 'go')
plt.hold(True)
plt.plot(x1_3,x2_3 , 'b.')
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
#</SOL>
```

```
/Users/jcid/anaconda/envs/mypy36/lib/python3.6/site-
packages/ipykernel_launcher.py:14: MatplotlibDeprecationWarning: pyplot.hold is
deprecated.
    Future behavior will be consistent with the long-time default:
    plot commands add elements without first clearing the
    Axes and/or Figure.

/Users/jcid/anaconda/envs/mypy36/lib/python3.6/site-
packages/matplotlib/__init__.py:917: UserWarning: axes.hold is deprecated.
Please remove it from your matplotlibrc and/or style files.
  warnings.warn(self.msg_depr_set % key)
/Users/jcid/anaconda/envs/mypy36/lib/python3.6/site-
packages/matplotlib/rcsetup.py:152: UserWarning: axes.hold is deprecated, will
be removed in 3.0
  warnings.warn("axes.hold is deprecated, will be removed in 3.0")
/Users/jcid/anaconda/envs/mypy36/lib/python3.6/site-
packages/ipykernel_launcher.py:16: MatplotlibDeprecationWarning: pyplot.hold is
deprecated.
    Future behavior will be consistent with the long-time default:
    plot commands add elements without first clearing the
    Axes and/or Figure.
  app.launch_new_instance()
```



According to this plot, which classes seem more difficult to distinguish?

1.4 4. Regression example

Now that we know how to load some data and visualize them, we will try to solve a simple regression task.

Our objective in this example is to predict the crime rates in different areas of the US using some socio-demographic data.

This dataset has 127 socioeconomic variables of different nature: categorical, integer, real, and for some of them there are also missing data (check wikipedia). This is usually a problem when training machine learning models, but we will ignore that problem and take only a small number of variables that we think can be useful for regression and which have no missing values.

5. population: population for community
6. householdsize: mean people per household
7. medIncome: median household income

The objective in the regresion problem is another real value that contains the *total number of violent crimes per 100K population*.

1.4.1 4.1 Loading the data

First of all, load the data from file *communities.csv* in a new array. This array should have 1994 rows (instances) and 128 columns.

[15]:
```
#<SOL>
# Open up the csv file in to a Python object
```

```
csv_file_object = csv.reader(open('communities.csv', 'r'))
datalist = []                    # Create a variable called 'data'.
for row in csv_file_object:      # Run through each row in the csv file,

    datalist.append(row)         # adding each row to the data variable


data_com = np.array(datalist)  # Then convert from a list to an array
                           # Be aware that each item is currently
                            # a string in this format
print(np.shape(data_com))
#</SOL>
```

```
(1994, 128)
```

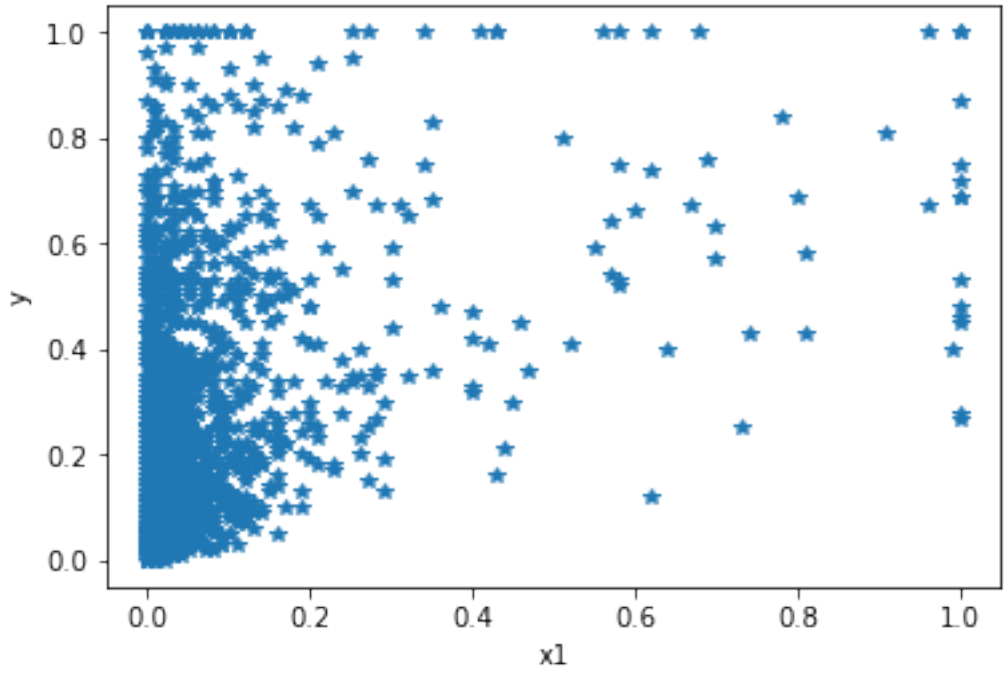
Take the columns (5,6,17) of the data and save them in a matrix *X_com*. This will be our input data. Convert this array into a *float* array. The shape should be (1994,3)

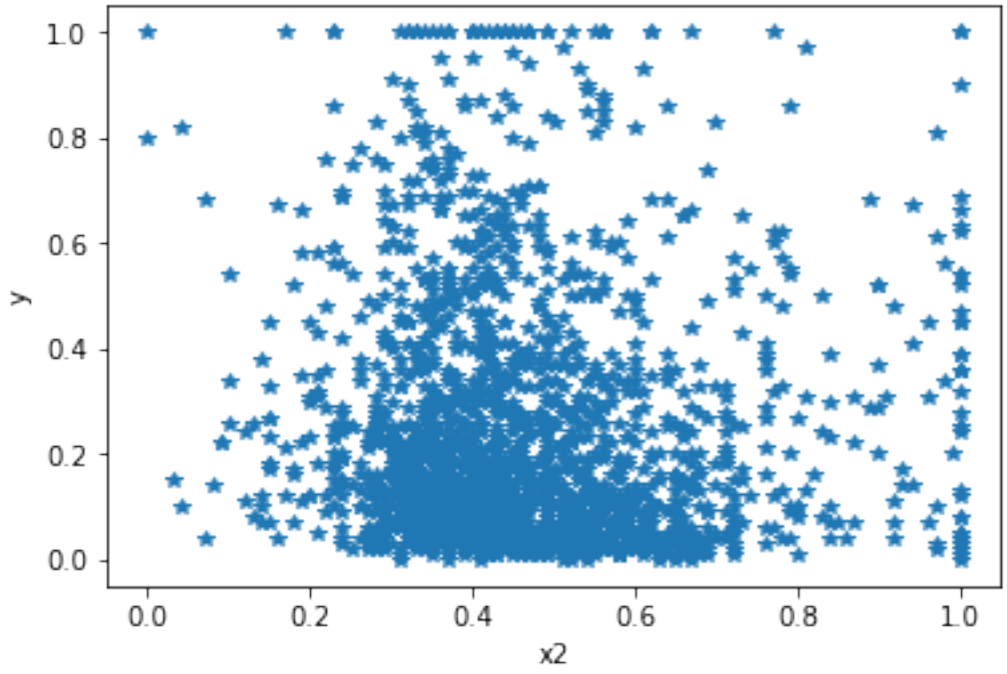
Get the last column of the data and save it in an array called *y_com*. Convert this matrix into a *float* array. Check that the shape is (1994,1) .

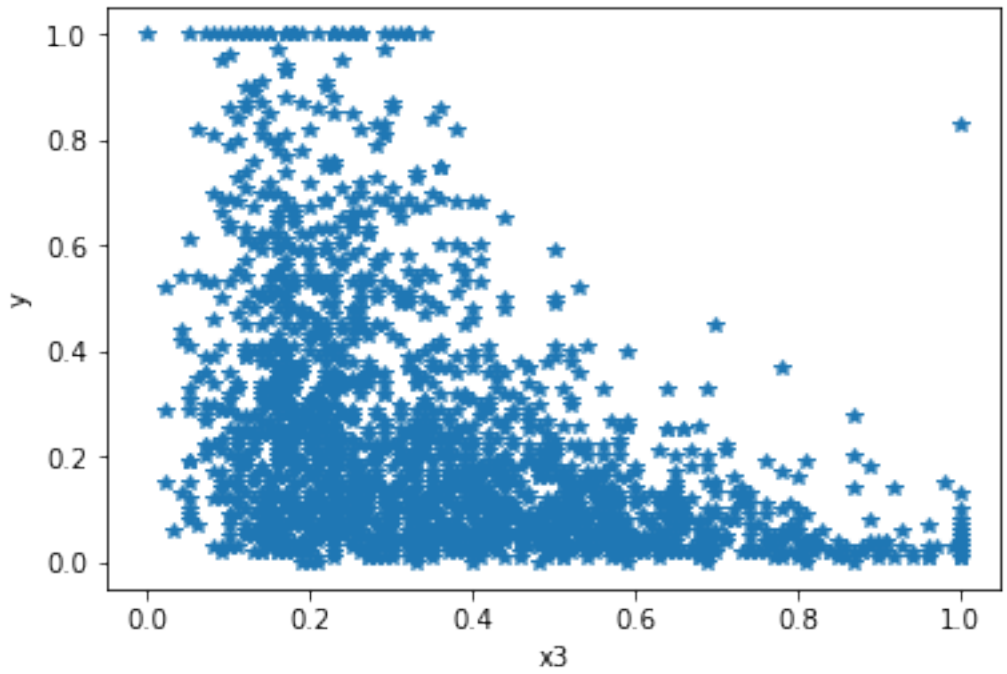
[16]:
```
#<SOL>
X_com = np.array(data_com[:,[5,6,17]],dtype= float)
y_com = np.array(data_com[:,-1,np.newaxis],dtype= float)
#</SOL>
```

Plot each variable in *X_com* versus *y_com* to have a first (partial) view of the data.

[17]:
```
#<SOL>
plt.figure(1)
plt.plot(X_com[:,0],y_com , '*')
plt.xlabel('x1')
plt.ylabel('y')
plt.figure(2)
plt.plot(X_com[:,1],y_com , '*')
plt.xlabel('x2')
plt.ylabel('y')
plt.figure(3)
plt.plot(X_com[:,2],y_com , '*')
plt.xlabel('x3')
plt.ylabel('y')
plt.show()
#</SOL>
```


1.0
0.8
0.6
0.4
0.2
0.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x1


1.0
0.8
0.6
0.4
0.2
0.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x2



1.4.2 4.2 Train/Test splitting

Now, we are about to start doing machine learning. But, first of all, we have to separate our data in train and test partitions.

The train data will be used to adjust the parameters (train) of our model. The test data will be used to evaluate our model.

Use *sklearn.cross_validation.train_test_split* to split the data in *train* (60%) and *test* (40%). Save the results in variables named *X_train*, *X_test*, *y_train*, *y_test*.

Important note In real applications, you would have no access to any targets for the test data. However, for illustratory purposes, when evaluating machine learning algorithms it is common to set aside a *test partition*, including the corresponding labels, so that you can use these targets to assess the performance of the method. When proceeding in this way, the test labels should never be used during the design. It is just allowed to use them as a final assessment step once the classifier or regression model has been fully adjusted.

[18]:
```
#<SOL>
X_train, X_test, y_train, y_test = train_test_split( X_com, y_com, test_size=0.4)
#</SOL>
```

1.4.3 4.3 Normalization

Most machine learning algorithms require that the data is standardized (mean=0, standard deviation= 1). Scikit-learn provides a tool to do that in the object *sklearn.preprocessing.StandardScaler* (but you can also try and program it by yourself, it easier than in MATLAB!!)

[19]:
```
#<SOL>
scaler = StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
print(X_train.mean())
print(X_train.std())
#</SOL>
```

```
5.57710696651e-16
1.0
```

1.4.4 4.4 Training

We will apply two different K-NN regressors for this example. One with K (*n_neighbors*) = 1 and the other with K=7.

Read the API and this example to understand how to fit the model.

[20]:
```
#<SOL>
print(X_train.shape)
print(y_train.shape)
knnreg = KNeighborsRegressor(n_neighbors=1)
knnreg.fit(X_train,y_train)

knnreg3 = KNeighborsRegressor(n_neighbors=7)
knnreg3.fit(X_train,y_train)
#</SOL>
```

```
(1196, 3)
(1196, 1)
```

[20]:
```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=7, p=2,
          weights='uniform')
```

1.4.5 4.5 Prediction and evaluation

Now use the two models you have trained to predict the test output *y_test*. To evaluate it measure the MSE.

The formula of MSE is

$$\text{MSE} = \frac{1}{K}\sum_{k=1}^{K}(\hat{y} - y)^2$$

[21]:
```
#<SOL>
y_pred = knnreg.predict(X_test)
print(np.mean((y_pred - y_test)**2))

y_pred3 = knnreg3.predict(X_test)
```

```
print(np.mean((y_pred3 - y_test)**2))
#</SOL>
```

```
0.071036716792
0.0380695437574
```

1.4.6 4.6 Saving the results

Finally we will save all our predictions for the model with K=1 in a csv file. To do so you can use the following code Snippet, where *y_pred* are the predicted output values for test.

[22]:
```
#<SOL>
y_pred = y_pred.squeeze()
csv_file_object = csv.writer(open('output.csv', 'w'))
for index, y_aux in enumerate(y_pred):      # Run through each row in the csv␣
 ↪file,
    csv_file_object.writerow([index,y_aux])
#</SOL>
```

[ ]:
```
```