Mansoura University
Faculty of Computer & Information
Information System Department

Data Mining

Lab Manual

4^{th} year

IS, IT, SWE, Bio

Dr. Amira Rezk

2019

Python for Data Science

How to Install Python

There are two ways to download and install Python

1. **Download Anaconda**. It comes with Python software along with preinstalled popular libraries.
2. Download **Python** from its official website. You have to manually install libraries.

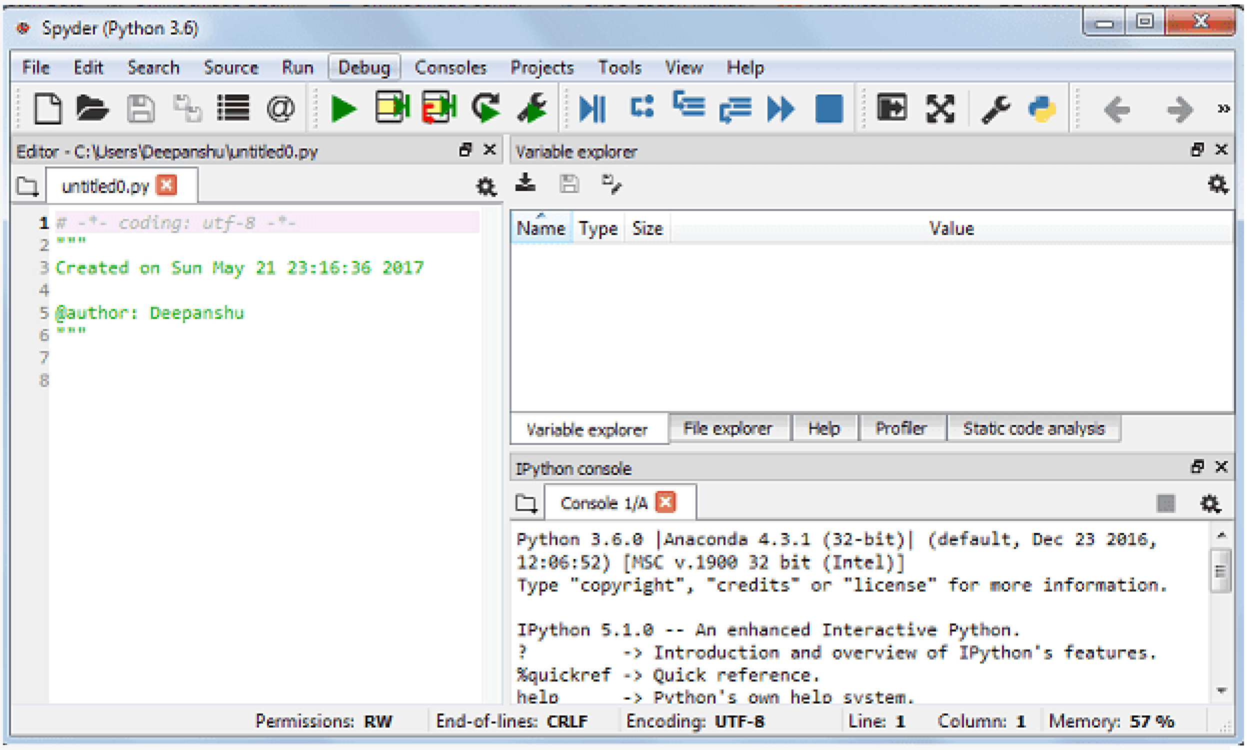
Recommended : *Go for first option and download anaconda. It saves a lot of time in learning and coding Python*

Coding Environments

Anaconda comes with two popular IDE :

1. Jupyter (Ipython) Notebook
2. Spyder

Spyder. It is like RStudio for Python. It gives an environment wherein writing python code is user-friendly. If you are a SAS User, you can think of it as SAS Enterprise Guide / SAS Studio. It comes with a syntax editor where you can write programs. It has a console to check each and every line of code. Under the 'Variable explorer', you can access your created data files and function. **I highly recommend Spyder!**



Spyder - Python Coding Environment

Jupyter (Ipython) Notebook

Jupyter is equivalent to markdown in R. It is useful when you need to present your work to others or when you need to create step by step project report as it can combine code, output, words, and graphics.

Spyder Shortcut Keys

The following is a list of some useful spyder shortcut keys which makes you more productive.

1. **Press F5** to run the entire script
2. **Press F9** to run selection or line
3. Press **Ctrl + 1** to comment / uncomment
4. Go to front of function and then **press Ctrl + I** to see documentation of the function
5. **Run** %reset -f to clean workspace
6. **Ctrl + Left click on object** to see source code
7. **Ctrl+Enter** executes the current cell.
8. **Shift+Enter** executes the current cell and advances the cursor to the next cell

List of arithmetic operators with examples

Arithmetic Operators	Operation	Example
+	Addition	10 + 2 = 12
–	Subtraction	10 – 2 = 8
*	Multiplication	10 * 2 = 20
/	Division	10 / 2 = 5.0
%	Modulus (Remainder)	10 % 3 = 1
**	Power	10 ** 2 = 100
//	Floor	17 // 3 = 5
(x + (d-1)) // d	Ceiling	(17 +(3-1)) // 3 = 6

Basic Programs

Example 1

```
#Basics
x = 10
y = 3
print("10 divided by 3 is", x/y)
print("remainder after 10 divided by 3 is", x%y)
```

Result :

10 divided by 3 is 3.33
remainder after 10 divided by 3 is 1

Example 2

```
x = 100
x > 80 and x <=95
x > 35 or x < 60
```

```
x > 80 and x <=95
Out[45]: False
x > 35 or x < 60
Out[46]: True
```

Comparison & Logical Operators	Description	Example
>	Greater than	5 > 3 returns True
<	Less than	5 < 3 returns False
>=	Greater than or equal to	5 >= 3 returns True
<=	Less than or equal to	5 <= 3 return False
==	Equal to	5 == 3 returns False
!=	Not equal to	5 != 3 returns True
and	Check both the conditions	x > 18 and x <=35
or	If atleast one condition hold True	x > 35 or x < 60
not	Opposite of Condition	not(x>7)

Assignment Operators

It is used to assign a value to the declared variable. For e.g. **x += 25 means x = x +25**.

```
x = 100
y = 10
x += y
print(x)
```

```
print(x)

110
```

In this case, x+=y implies x=x+y which is x = 100 + 10.

Similarly, *you can use x-=y, x*=y and x /=y*

Python Data Structure

In every programming language, it is important to understand the data structures. Following are some data structures used in Python.

1. List

It is a sequence of multiple values. It allows us to store different types of data such as integer, float, string etc. See the examples of list below. First one is an integer list containing only integer. Second one is string list containing only string values. Third one is mixed list containing integer, string and float values.

1. x = [1, 2, 3, 4, 5]
2. y = [‘A’, ‘O’, ‘G’, ‘M’]
3. z = [‘A’, 4, 5.1, ‘M’]

Get List Item

We can extract list item using Indexes. **Index starts from 0 and end with (number of elements-1).**

x = [1, 2, 3, 4, 5]
x[0]
x[1]
x[4]
x[-1]
x[-2]

```
x[0]
Out[68]: 1

x[1]
Out[69]: 2

x[4]
Out[70]: 5

x[-1]
Out[71]: 5

x[-2]
Out[72]: 4
```

x[0] picks first element from list. **Negative sign** tells Python to search list item from right to left. **x[-1]** selects the last element from list.

You can select multiple elements from a list using the following method

x[:3] returns [1, 2, 3]

2. Tuple

A tuple is similar to a list in the sense that it is a sequence of elements. The difference between list and tuple are as follows :

1. A tuple cannot be changed once constructed whereas list can be modified.
2. A tuple is created by placing comma-separated values inside parentheses **( )**. Whereas, list is created inside square brackets **[ ]**

Examples

```
K = (1,2,3)
State = ('Delhi','Maharashtra','Karnataka')
```

Perform for loop on Tuple

```
for i in State:
   print(i)
```

```
Delhi

Maharashtra

Karnataka
```

Functions

Like **print()**, you can create your own custom function. It is also called user-defined functions. It helps you in automating the repetitive task and calling reusable code in easier way.

Rules to define a function

1. Function starts with **def** keyword followed by function name and **( )**
2. Function body starts with a colon (:) and is indented
3. The keyword **return** ends a function and give value of previous expression.

```
def sum_fun(a, b):
  result = a + b
  return result

z = sum_fun(10, 15)
```

Result : z = 25

Suppose you want python to assume **0 as default value** if no value is specified for parameter b.

```
def sum_fun(a, b=0):
  result = a + b
  return result
z = sum_fun(10)
```

In the above function, b is set to be 0 if no value is provided for parameter b. It does not mean no other value than 0 can be set here. It can also be used as **z = sum_fun(10, 15)**

Conditional Statements (if else)

Conditional statements are commonly used in coding. It is IF ELSE statements. It can be read like : " if a condition holds true, then execute something. Else execute something else"

Note : The if and else statements ends with a colon **:**

Example

```
k = 27
if k%5 == 0:
 print('Multiple of 5')
else:
 print('Not a Multiple of 5')
```

Result : Not a Multiple of 5

Popular python packages for Data Analysis & Visualization

Some of the leading packages in Python along with equivalent libraries in R are as follows:

1. **pandas**. For data manipulation and data wrangling. A collections of functions to understand and explore data. It is counterpart of **dplyr** and **reshape2** packages in R.

2. **NumPy**. For numerical computing. It's a package for efficient array computations. It allows us to do some operations on an entire column or table in one line. It is roughly approximate to **Rcpp** package in R which eliminates the limitation of slow speed in R. **Numpy Tutorial**

3. **Scipy**. For mathematical and scientific functions such as integration, interpolation, signal processing, linear algebra, statistics, etc. It is built on Numpy.

4. **Scikit-learn.** A collection of machine learning algorithms. It is built on Numpy and Scipy. It can perform all the techniques that can be done in R using **glm, knn, randomForest, rpart, e1071** packages.

5. **Matplotlib.** For data visualization. It's a leading package for graphics in Python. It is equivalent to **ggplot2** package in R.

6. **Statsmodels.** For statistical and predictive modeling. It includes various functions to explore data and generate descriptive and predictive analytics. It allows users to run descriptive statistics, methods to impute missing values, statistical tests and take table output to HTML format.

7. **pandasql.** It allows SQL users to write SQL queries in Python. It is very helpful for people who loves writing SQL queries to manipulate data. It is equivalent to **sqldf** package in R.

Maximum of the above packages are already preinstalled in Spyder.

Popular Python Commands

The commands below would help you to install and update new and existing packages. Let's say, you want to install / uninstall **pandas** package.

Install Package
!pip install pandas

Uninstall Package
!pip uninstall pandas

Show Information about Installed Package
!pip show pandas

List of Installed Packages
!pip list

Upgrade a package
!pip install --upgrade pandas

How to import a package

There are multiple ways to import a package in Python. It is important to understand the difference between these styles.

1. import pandas as pd

It imports the package pandas under the alias pd. A function DataFrame in package pandas is then submitted with pd.DataFrame.

2. import pandas

It imports the package without using alias but here the function **DataFrame** is submitted with full package name **pandas.DataFrame**

3. from pandas import *
It imports the whole package and the function **DataFrame** is executed simply by typing **DataFrame.** It sometimes creates confusion when same function name exists in more than one package.

Pandas Data Structures: Series and DataFrame

In pandas package, there are two data structures - series and dataframe. These structures are explained below in detail :

1- Series is a one-dimensional array. You can access individual elements of a series using position. It's similar to vector in R.
In the example below, we are generating 5 random values.

```
import pandas as pd
import numpy as np
s1 = pd.Series(np.random.randn(5))
s1
```

```
0   -2.412015
1   -0.451752
2    1.174207
3    0.766348
4   -0.361815
dtype: float64
```

Extract first and second value

You can get a particular element of a series using index value. See the examples below -

s1[0]

```
-2.412015
```

s1[1]

```
-0.451752
```

s1[:3]

```
0   -2.412015
1   -0.451752
2    1.174207
```

2. DataFrame

It is equivalent to data.frame in R. It is a 2-dimensional data structure that can store data of different data types such as characters, integers, floating point values, factors. Those who are well-conversant with MS Excel, they can think of data frame as Excel Spreadsheet.

Comparison of Data Type in Python and Pandas

The following table shows how Python and pandas package stores data.

Data Type	Pandas	Standard Python
For character variable	object	string
For categorical variable	category	-
For Numeric variable without decimals	int64	int
Numeric characters with decimals	float64	float
For date time variables	datetime64	-

Important Pandas Functions

The table below shows comparison of pandas functions with R functions for various data wrangling and manipulation tasks. It would help you to memorize pandas functions. It's a very handy information for programmers who are new to Python. It includes solutions for most of the frequently used data exploration tasks.

Functions	R	Python (pandas package)
Installing a package	install.packages('name')	!pip install name
Loading a package	library(name)	import name as other_name
Checking working directory	getwd()	import os os.getcwd()

Setting working directory	setwd()	os.chdir()
List files in a directory	dir()	os.listdir()
Remove an object	rm('name')	del object
Select Variables	select(df, x1, x2)	df[['x1', 'x2']]
Drop Variables	select(df, -(x1:x2))	df.drop(['x1', 'x2'], axis = 1)
Filter Data	filter(df, x1 >= 100)	df.query('x1 >= 100')
Structure of a DataFrame	str(df)	df.info()
Summarize dataframe	summary(df)	df.describe()
Get row names of dataframe "df"	rownames(df)	df.index
Get column names	colnames(df)	df.columns
View Top N rows	head(df,N)	df.head(N)
View Bottom N rows	tail(df,N)	df.tail(N)
Get dimension of data frame	dim(df)	df.shape
Get number of rows	nrow(df)	df.shape[0]
Get number of columns	ncol(df)	df.shape[1]
Length of data frame	length(df)	len(df)
Get random 3 rows from dataframe	sample_n(df, 3)	df.sample(n=3)
Get random 10% rows	sample_frac(df, 0.1)	df.sample(frac=0.1)
Check Missing Values	is.na(df$x)	pd.isnull(df.x)
Sorting	arrange(df, x1, x2)	df.sort_values(['x1', 'x2'])
Rename Variables	rename(df, newvar = x1)	df.rename(columns={'x1': 'newvar'})

IMPORTING DATA INTO PYTHON

While importing external files, we need to check the following points -

1. Check whether header row exists or not
2. Treatment of special values as missing values
3. Consistent data type in a variable (column)
4. Date Type variable in consistent date format.
5. No truncation of rows while reading external data

1. Import CSV files

It is important to note that **a single backslash does not work** when specifying the file path**.** You need to either **change it to forward slash or add one more backslash** like below

```
import pandas as pd
mydata= pd.read_csv("C:\\Users\\Deepanshu\\Documents\\file1.csv")
```

If no header (title) in raw data file

```
mydata1  = pd.read_csv("C:\\Users\\Deepanshu\\Documents\\file1.csv", header = None)
```

You need to include **header = None** option to tell Python there is no column name (header) in data.

Add Column Names

We can include column names by using names= option.

```
mydata2  = pd.read_csv("C:\\Users\\Deepanshu\\Documents\\file1.csv", header =
None, names = ['ID', 'first_name', 'salary'])
```

The variable names can also be added separately by using the following command.

```
mydata1.columns = ['ID', 'first_name', 'salary']
```

2. Import File from URL

You don't need to perform additional steps to fetch data from URL. Simply put **URL** in read_csv() function (applicable only for CSV files stored in URL).

```
mydata  = pd.read_csv("http://winterolympicsmedals.com/medals.csv")
```

3. Read Text File

We can use read_table() function to pull data from text file. We can also use read_csv() with sep= "\t" to read data from tab-separated file.

```
mydata = pd.read_table("C:\\Users\\Deepanshu\\Desktop\\example2.txt")
mydata  = pd.read_csv("C:\\Users\\Deepanshu\\Desktop\\example2.txt", sep ="\t")
```

4. Read Excel File

The read_excel() function can be used to import excel data into Python.

```
mydata = pd.read_excel("https://www.eia.gov/dnav/pet/hist_xls/RBRTEd.xls",sheetname="Data 1", skiprows=2)
```

*If you do not specify name of sheet in **sheetname= option**, it would take by default first sheet.*

5. Read delimited file

Suppose you need to import a file that is separated with white spaces.

```
mydata2 = pd.read_table("http://www.ssc.wisc.edu/~bhansen/econometrics/invest.dat", sep="\s+", header = None)
```

To include variable names, use the names= option like below:

```
mydata3 = pd.read_table("http://www.ssc.wisc.edu/~bhansen/econometrics/invest.dat", sep="\s+", names=['a', 'b', 'c', 'd'])
```

6. Read SAS File

We can import SAS data file by using read_sas() function.

```
mydata4 = pd.read_sas('cars.sas7bdat')
```

7. Read SQL Table

We can extract table from SQL database (Teradata / SQL Server). See the program below -

```
import sqlite3
from pandas.io import sql
conn = sqlite3.connect('C:/Users/Deepanshu/Downloads/flight.db')
query = "SELECT * FROM flight;"
results = pd.read_sql(query, con=conn)
print results.head()
```

8. Read sample of rows and columns

By specifying nrows= and usecols=, you can fetch specified number of rows and columns.

```
mydata7  = pd.read_csv("http://winterolympicsmedals.com/medals.csv", nrows=5, usecols=(1,5,7))
```

nrows = 5 implies you want to import only first 5 rows and usecols= refers to specified columns you want to import.

9. Skip rows while importing

Suppose you want to skip first 5 rows and wants to read data from 6th row (6th row would be a header row)

```
mydata8  = pd.read_csv("http://winterolympicsmedals.com/medals.csv", skiprows=5)
```

10. Specify values as missing values

By including na_values= option, you can specify values as missing values. In this case, we are telling python to consider dot (.) as missing cases.

```
mydata9  = pd.read_csv("workingfile.csv", na_values=['.'])
```

Data Manipulation with pandas

1. Import Required Packages

You can import required packages using **import** statement. In the syntax below, we are asking Python to import numpy and pandas package. The 'as' is used to alias package name.

```
import numpy as np
import pandas as pd
```

2. Build DataFrame

We can build dataframe using **DataFrame()** function of pandas package.

```
mydata = {'productcode': ['AA', 'AA', 'AA', 'BB', 'BB', 'BB'],
        'sales': [1010, 1025.2, 1404.2, 1251.7, 1160, 1604.8],
        'cost' : [1020, 1625.2, 1204, 1003.7, 1020, 1124]}
df = pd.DataFrame(mydata)
```

In this dataframe, we have three variables - productcode, sales, cost.

productcode	sales	cost
AA	1010	1020
AA	1025.2	1625.2
AA	1404.2	1204
BB	1251.7	1003.7
BB	1160	1020
BB	1604.8	1124

Sample DataFrame

To import data from CSV file

You can use read_csv() function from pandas package to get data into python from CSV file.

```
mydata= pd.read_csv("C:\\Users\\Deepanshu\\Documents\\file1.csv")
```

Make sure you use **double backslash** when specifying path of CSV file. Alternatively, you can use forward slash to mention file path inside read_csv() function.

3. To see number of rows and columns

You can run the command below to find out number of rows and columns.

```
df.shape
```

Result : (6, 3). It means 6 rows and 3 columns.

4. To view first 3 rows

The df.head(N) function can be used to check out first some N rows.

```
df.head(3)
```

```
     cost productcode   sales
0  1020.0          AA  1010.0
1  1625.2          AA  1025.2
2  1204.0          AA  1404.2
```

5. Select or Drop Variables

To keep a single variable, you can write in any of the following three methods -

```
df.productcode
df["productcode"]
df.loc[: , "productcode"]
```

To select variable by column position, you can use **df.iloc** function. In the example below, we are selecting second column. Column Index starts from 0. Hence, 1 refers to second column.

```
df.iloc[: , 1]
```

We can keep multiple variables by specifying desired variables inside [ ]. Also, we can make use of df.loc() function.

```
df[["productcode", "cost"]]
df.loc[ : , ["productcode", "cost"]]
```

Drop Variable

We can remove variables by using df.drop() function. See the example below:

```
df2 = df.drop(['sales'], axis = 1)
```

6. To summarize data frame

To summarize or explore data, you can submit the command below.
df.describe()

```
              cost       sales
count     6.000000     6.00000
mean   1166.150000  1242.65000
std     237.926793   230.46669
min    1003.700000  1010.00000
25%    1020.000000  1058.90000
50%    1072.000000  1205.85000
75%    1184.000000  1366.07500
max    1625.200000  1604.80000
```

To summaries all the **character variables**, you can use the following script.

```
df.describe(include=['O'])
```

Similarly, you can use **df.describe(include=['float64'])** to view summary of all the numeric variables with decimals.

To select only a particular variable, you can write the following code -

```
df.productcode.describe()
OR
df["productcode"].describe()
```

```
count      6
unique     2
top       BB
freq       3
Name: productcode, dtype: object
```

7. To calculate summary statistics

We can manually find out summary statistics such as count, mean, median by using commands below

```
df.sales.mean()
df.sales.median()
df.sales.count()
df.sales.min()
df.sales.max()
```

8. Filter Data

Suppose you are asked to apply condition - productcode is equal to "AA" and sales greater than or equal to 1250.

```
df1 = df[(df.productcode == "AA") & (df.sales >= 1250)]
```

It can also be written like :

```
df1 = df.query('(productcode == "AA") & (sales >= 1250)')
```

In the second query, we **do not need** to specify DataFrame along with variable name.

9. Sort Data

In the code below, we are arrange data in ascending order by sales.

```
df.sort_values(['sales'])
```

10. Group By : Summary by Grouping Variable

Like SQL GROUP BY, you want to summarize continuous variable by classification variable. In this case, we are calculating average sale and cost by product code.

```
df.groupby(df.productcode).mean()
```

```
                  cost        sales

productcode

AA          1283.066667  1146.466667

BB          1049.233333  1338.833333
```

Instead of summarising for multiple variable, you can run it for a single variable i.e. sales. Submit the following script.

```
df["sales"].groupby(df.productcode).mean()
```

11. Define Categorical Variable

Let's create a classification variable - id which contains only 3 unique values - 1/2/3.

```
df0 = pd.DataFrame({'id': [1, 1, 2, 3, 1, 2, 2]})
```

Let's define as a categorical variable.

We can use astype() function to make id as a categorical variable.

```
df0.id = df0["id"].astype('category')
```

Summarize this classification variable to check descriptive statistics.

```
df0.describe()
```

```
        id

count    7

unique   3

top      2

freq     3
```

Frequency Distribution

You can calculate frequency distribution of a categorical variable. It is one of the method to explore a categorical variable.

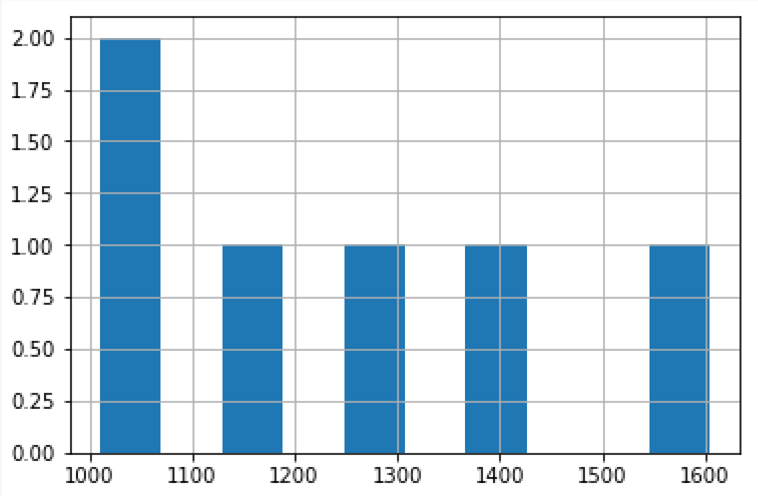
```
df['productcode'].value_counts()
```

```
BB    3

AA    3
```

12. Generate Histogram

Histogram is one of the method to check distribution of a continuous variable. In the figure shown below, there are two values for variable 'sales' in range 1000-1100. In the remaining intervals, there is only a single value. In this case, there are only 5 values. If you have a large dataset, you can plot histogram to identify outliers in a continuous variable.

```
df['sales'].hist()
```

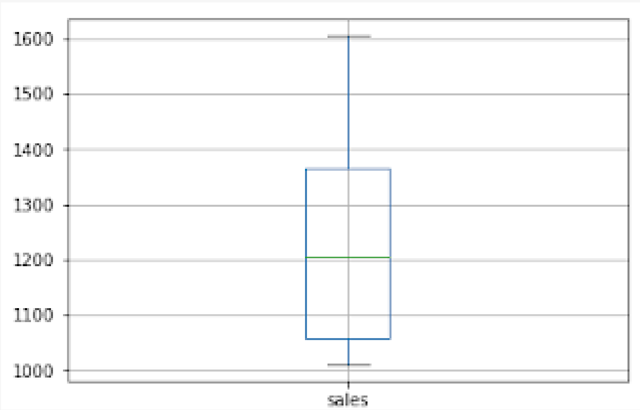


Histogram

13. BoxPlot

Boxplot is a method to visualize continuous or numeric variable. It shows minimum, Q1, Q2, Q3, IQR, maximum value in a single graph.

```
df.boxplot(column='sales')
```



BoxPlot

Data Science using Python - Examples

In this section, we cover how to perform data mining and machine learning algorithms with Python. sklearn is the most frequently used library for running data mining and machine learning algorithms. We will also cover statsmodels library for regression techniques. statsmodels library generates formattable output which can be used further in project report and presentation.

1. Install the required libraries

Import the following libraries before reading or exploring data

```
#Import required libraries
import pandas as pd
import statsmodels.api as sm
import numpy as np
```

2. Download and import data into Python

With the use of python library, we can easily get data from web into python.

```
# Read data from web
df = pd.read_csv("https://stats.idre.ucla.edu/stat/data/binary.csv")
```

Variables Type Description

```
gre Continuous Graduate Record Exam score
gpa Continuous Grade Point Average
rank Categorical Prestige of the undergraduate institution
admit Binary Admission in graduate school
```

The binary variable admit is a target variable.

3. Explore Data

Let's explore data. We'll answer the following questions:

1. How many rows and columns in the data file?
2. What are the distribution of variables?
3. Check if any outlier(s)
4. If outlier(s), treat them
5. Check if any missing value(s)
6. Impute Missing values (if any)

```
# See no. of rows and columns
df.shape
```

Result : 400 rows and 4 columns

In the code below, we rename the variable rank to '**position**' as rank is already a function in python.

```
# rename rank column
df = df.rename(columns={'rank': 'position'})
```

Summarize and plot all the columns.

```
# Summarize
df.describe()
# plot all of the columns
df.hist()
```

Categorical variable Analysis

It is important to check the frequency distribution of categorical variable. It helps to answer the question whether data is skewed.

```
# Summarize
df.position.value_counts(ascending=True)
```

```
1     61
4     67
3    121
2    151
```

Generating Crosstab

By looking at cross tabulation report, we can check whether we have enough number of events against each unique values of categorical variable.

```
pd.crosstab(df['admit'], df['position'])
```

```
position   1   2   3   4

admit

0         28  97  93  55

1         33  54  28  12
```

Number of Missing Values

We can write a simple loop to figure out the number of blank values in all variables in a dataset.

```
for i in list(df.columns) :
   k = sum(pd.isnull(df[i]))
   print(i, k)
```

In this case, there are no missing values in the dataset.

4. Logistic Regression Model

Logistic Regression is a special type of regression where target variable is categorical in nature and independent variables be discrete or continuous. In this post, we will demonstrate only **binary logistic regression** which takes only binary values in target variable. Unlike linear regression, logistic regression model returns probability of target variable.It assumes binomial distribution of dependent variable. In other words, it belongs to binomial family.

In python, we can write R-style model formula **y ~ x1 + x2 + x3** using **patsy** and **statsmodels** libraries. In the formula, we need to define variable 'position' as a categorical variable by mentioning it inside capital C(). You can also define reference category using **reference=** option.

```
#Reference Category
from patsy import dmatrices, Treatment
y, X = dmatrices('admit ~ gre + gpa + C(position, Treatment(reference=4))', df,
return_type = 'dataframe')
```

It returns two datasets - X and y. The dataset 'y' contains variable admit which is a target variable. The other dataset 'X' contains Intercept (constant value), dummy variables for Treatment, gre and gpa. Since 4 is set as a reference category, it will be 0 against all the three dummy variables. See sample below :

```
P  P1 P2 P3

3  0  0  1

3  0  0  1

1  1  0  0

4  0  0  0

4  0  0  0

2  0  1  0
```

Split Data into two parts

80% of data goes to training dataset which is used for building model and 20% goes to test dataset which would be used for validating the model.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

Build Logistic Regression Model

By default, the **regression without formula style** does not include intercept. To include it, we already have added **intercept** in **X_train** which would be used as a predictor.

```
#Fit Logit model
logit = sm.Logit(y_train, X_train)
result = logit.fit()


#Summary of Logistic regression model
result.summary()
result.params
```

```
                          Logit Regression Results
==============================================================
Dep. Variable:    admit   No. Observations:    320
Model:            Logit   Df Residuals:        315
Method:             MLE   Df Model:              4
Date:  Sat, 20 May 2017   Pseudo R-squ.:    0.03399
Time:          19:57:24    Log-Likelihood:  -193.49
converged:         True   LL-Null:          -200.30
LLR p-value:  0.008627
==============================================================
coef    std err         z       P|z|      [95.0% Conf. Int.]
--------------------------------------------------------------
```

```
C(position)[T.1]     1.4933       0.440       3.392       0.001
0.630     2.356

C(position)[T.2]     0.6771       0.373       1.813       0.070
-0.055     1.409

C(position)[T.3]     0.1071       0.410       0.261       0.794
-0.696     0.910

gre                  0.0005       0.001       0.442       0.659
-0.002     0.003

gpa                  0.4613       0.214      -2.152       0.031
-0.881    -0.041

==============================================================
```

Confusion Matrix and Odd Ratio

Odd ratio is exponential value of parameter estimates.

```
#Confusion Matrix
result.pred_table()
#Odd Ratio
np.exp(result.params)
```

Prediction on Test Data

In this step, we take estimates of logit model which was built on training data and then later apply it into test data.

```
#prediction on test data
y_pred = result.predict(X_test)
```

Calculate Area under Curve (ROC)

```
# AUC on test data
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred)
auc(false_positive_rate, true_positive_rate)
```

Result : AUC = 0.6763

Calculate Accuracy Score

```
accuracy_score([ 1 if p > 0.5 else 0 for p in y_pred ], y_test)
```

Decision Tree Model

Decision trees can have a target variable continuous or categorical. When it is continuous, it is called regression tree. And when it is categorical, it is called classification tree. It selects a variable at each step that best splits the set of values. There are several algorithms to find best split. Some of them are Gini, Entropy, C4.5, Chi-Square. There are several advantages of decision tree. It is simple to use and easy to understand. It requires a very few data preparation steps. It can handle mixed data - both categorical and continuous variables. In terms of speed, it is a very fast algorithm.

```
#Drop Intercept from predictors for tree algorithms

X_train = X_train.drop(['Intercept'], axis = 1)

X_test = X_test.drop(['Intercept'], axis = 1)

#Decision Tree

from sklearn.tree import DecisionTreeClassifier

model_tree = DecisionTreeClassifier(max_depth=7)

#Fit the model:

model_tree.fit(X_train,y_train)

#Make predictions on test set

predictions_tree = model_tree.predict_proba(X_test)

  #AUC

false_positive_rate, true_positive_rate, thresholds =
roc_curve(y_test, predictions_tree[:,1])

auc(false_positive_rate, true_positive_rate)
```

Result : AUC = 0.664

Important Note

Feature engineering plays an important role in building predictive models. In the above case, we have not performed variable selection. We can also select best parameters by using grid search fine tuning technique.

Random Forest Model

Decision Tree has limitation of overfitting which implies it does not generalize pattern. It is very sensitive to a small change in training data. To overcome this problem, random forest comes into picture. It grows a large number of trees on randomised data. It selects random number of variables to grow each tree. It is more robust algorithm than decision tree. It is one of the most popular machine learning algorithm. It is commonly used in data science competitions. It is always ranked in top 5 algorithms. It has become a part of every data science toolkit.

```
#Random Forest

from sklearn.ensemble import RandomForestClassifier

model_rf = RandomForestClassifier(n_estimators=100,
max_depth=7)



#Fit the model:

target = y_train['admit']

model_rf.fit(X_train,target)



#Make predictions on test set

predictions_rf = model_rf.predict_proba(X_test)
```

```
#AUC

false_positive_rate, true_positive_rate, thresholds =
roc_curve(y_test, predictions_rf[:,1])

auc(false_positive_rate, true_positive_rate)



#Variable Importance

importances = pd.Series(model_rf.feature_importances_,
index=X_train.columns).sort_values(ascending=False)

print(importances)

importances.plot.bar()
```

Result : AUC = 0.6974

Grid Search - Hyper Parameters Tuning

The sklearn library makes hyper-parameters tuning very easy. It is a strategy to select the best parameters for an algorithm. In scikit-learn they are passed as arguments to the constructor of the estimator classes. For example, max_features in randomforest. alpha for lasso.

```
from sklearn.model_selection import GridSearchCV

rf = RandomForestClassifier()

target = y_train['admit']



param_grid = {

    'n_estimators': [100, 200, 300],

    'max_features': ['sqrt', 3, 4]

}
```

```
CV_rfc = GridSearchCV(estimator=rf , param_grid=param_grid,
cv= 5, scoring='roc_auc')

CV_rfc.fit(X_train,target)



#Parameters with Scores

CV_rfc.grid_scores_

#Best Parameters

CV_rfc.best_params_

CV_rfc.best_estimator_

#Make predictions on test set

predictions_rf = CV_rfc.predict_proba(X_test)



#AUC

false_positive_rate, true_positive_rate, thresholds =
roc_curve(y_test, predictions_rf[:,1])

auc(false_positive_rate, true_positive_rate)
```

Cross Validation

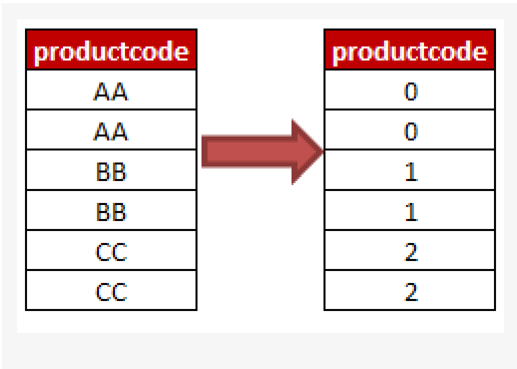
```
# Cross Validation
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_predict,cross_val_score
target = y['admit']
prediction_logit = cross_val_predict(LogisticRegression(), X, target, cv=10,
method='predict_proba')
#AUC
cross_val_score(LogisticRegression(fit_intercept = False), X, target, cv=10,
scoring='roc_auc')
```

Data Mining : PreProcessing Steps

1. The machine learning package sklearn requires all categorical variables in numeric form.

Hence, we need to convert all character/categorical variables to be numeric. This can be accomplished using the following script. In sklearn, there is already a function for this step.

```
from sklearn.preprocessing import LabelEncoder

def ConverttoNumeric(df):

    cols =
list(df.select_dtypes(include=['category','object']))

    le = LabelEncoder()

    for i in cols:

        try:

            df[i] = le.fit_transform(df[i])

        except:

            print('Error in Variable :'+i)

    return df

ConverttoNumeric(df)
```



2. Create Dummy Variables

Suppose you want to convert categorical variables into dummy variables. It is different to the previous example as it creates dummy variables instead of convert it in numeric form.

```
productcode_dummy = pd.get_dummies(df["productcode"])
df2 = pd.concat([df, productcode_dummy], axis=1)
```

The output looks like below -

	AA	BB
0	1	0
1	1	0
2	1	0
3	0	1
4	0	1
5	0	1

Create k-1 Categories

To avoid multi-collinearity, you can set one of the category as reference category and leave it while creating dummy variables. In the script below, we are leaving first category.

```
productcode_dummy = pd.get_dummies(df["productcode"], prefix='pcode',
drop_first=True)
df2 = pd.concat([df, productcode_dummy], axis=1)
```

3. Impute Missing Values

Imputing missing values is an important step of predictive modeling. In many algorithms, if missing values are not filled, it removes complete row. If data contains a lot of missing values, it can lead to huge data loss. There are multiple ways to impute missing values. Some of the common techniques - to replace missing value with mean/median/zero. It makes sense to replace missing value with 0 when 0 signifies meaningful. For example, whether customer holds a credit card product.

Fill missing values of a particular variable

```
# fill missing values with 0
df['var1'] = df['var1'].fillna(0)
# fill missing values with mean
df['var1'] = df['var1'].fillna(df['var1'].mean())
```

Apply imputation to the whole dataset

```
from sklearn.preprocessing import Imputer

# Set an imputer object
mean_imputer = Imputer(missing_values='NaN', strategy='mean', axis=0)

# Train the imputor
mean_imputer = mean_imputer.fit(df)

# Apply imputation
df_new = mean_imputer.transform(df.values)
```

4. Outlier Treatment

There are many ways to handle or treat outliers (or extreme values). Some of the methods are as follows -

1. Cap extreme values at 95th / 99th percentile depending on distribution
2. Apply log transformation of variables. See below the implementation of log transformation in Python.

```
import numpy as np
df['var1'] = np.log(df['var1'])
```

5. Standardization

In some algorithms, it is required to standardize variables before running the actual algorithm. Standardization refers to the process of making mean of variable zero and unit variance (standard deviation).

```
#load dataset
dataset = load_boston()
predictors = dataset.data
target = dataset.target
df = pd.DataFrame(predictors, columns = dataset.feature_names)

#Apply Standardization
from sklearn.preprocessing import StandardScaler
k = StandardScaler()
df2 = k.fit_transform(df)
```

Creating a Clustering Model in Python

1: Exploratory Data Analysis

You will need to install a few modules, including one new module called **Sci-kit Learn** – a collection of tools for machine learning and data mining in Python

Cluster is the sci-kit module that imports functions with clustering algorithms, hence why it is imported from sci-kit.

First, let's import all necessary modules into our iPython Notebook and do some exploratory data analysis.

```
In [18]:

import pandas as pd

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn

from sklearn import cluster

%matplotlib inline

faithful = pd.read_csv('/Users/michaelrundell/Desktop/faithful.csv')

faithful.head()

Out[18]:
```

	eruptions	waiting
0	3.600	79
1	1.800	54
2	3.333	74
3	2.283	62
4	4.533	85

Reading the old faithful csv and importing all necessary values

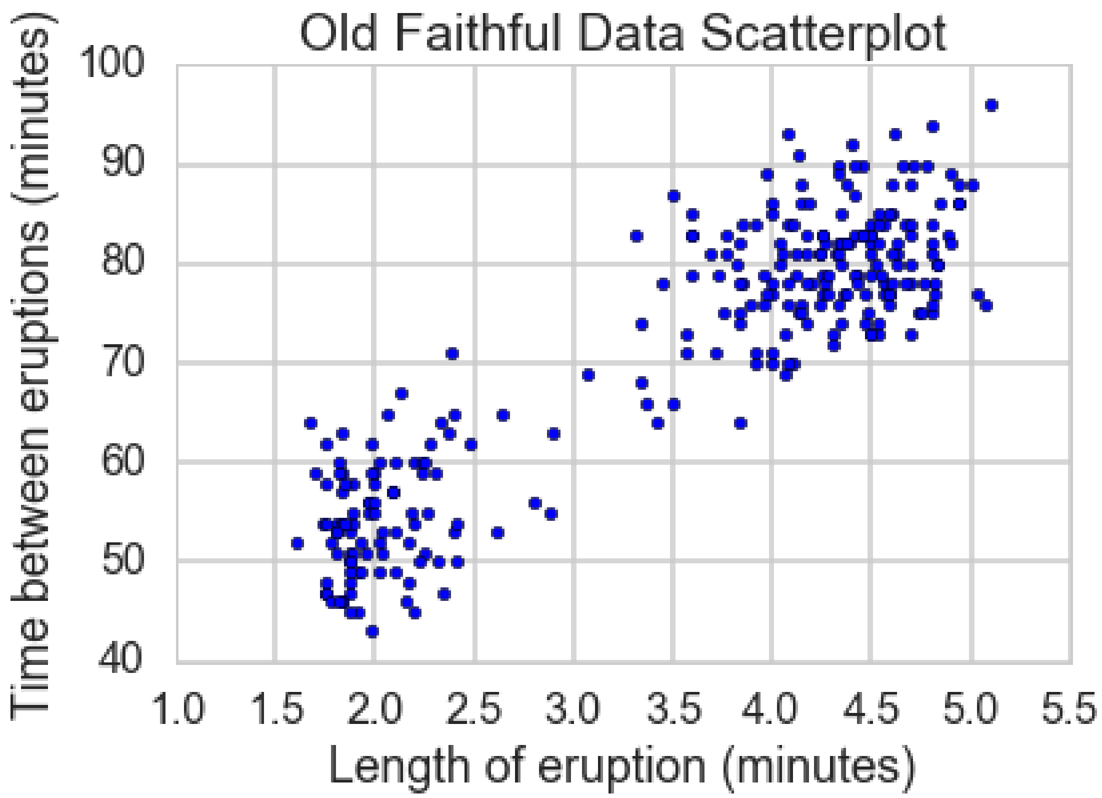
All I've done is read the csv from my local directory, which happens to be my computer's desktop, and shown the first 5 entries of the data. Fortunately, I know this data set has no columns with missing or NaN values, so we can skip the data cleaning section in this example. Let's take a look at a basic scatterplot of the data.

In [19]:

```
faithful.columns = ['eruptions', 'waiting']



plt.scatter(faithful.eruptions, faithful.waiting)

plt.title('Old Faithful Data Scatterplot')

plt.xlabel('Length of eruption (minutes)')

plt.ylabel('Time between eruptions (minutes)')
```

Out[19]:

```
<matplotlib.text.Text at 0x12a29bba8>
```



Renaming the columns and using matplotlib to create a simple scatterplot.

Some quick notes on my process here: I renamed the columns – they don't look any different to the naked eye, but the "waiting" column had an extra space before the word, and to prevent any confusion with further analysis I changed it to ensure I don't forget or make any mistakes down the road.

2: Building the cluster model

What we see is a scatter plot that has two clusters that are easily apparent, but the data set does not label any observation as belonging to either group. The next few steps will cover the process of visually differentiating the two groups. In the code below, I establish some important variables and alter the format of the data.

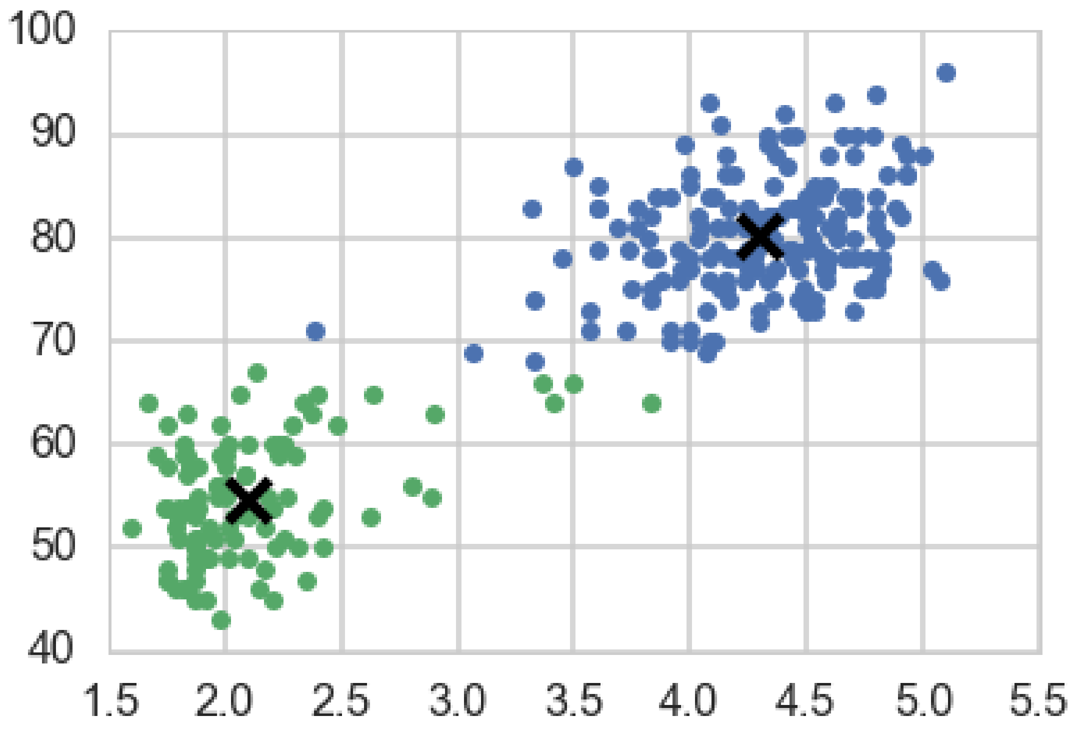
```
In [20]:

faith = np.array(faithful)



k = 2

kmeans = cluster.KMeans(n_clusters=k)

kmeans.fit(faith)



labels = kmeans.labels_

centroids = kmeans.cluster_centers_
```

Formatting and function creation.

1. I read the faithful dataframe as a numpy array in order for sci-kit to be able to read the data.
2. K = 2 was chosen as the number of clusters because there are 2 clear groupings we are trying to create.
3. The 'kmeans' variable is defined by the output called from the cluster module in sci-kit. We have it take on a K number of clusters, and fit the data in the array 'faith'.

Now that we have set up the variables for creating a cluster model, let's create a visualization. The code below will plot a scatter plot that colors by cluster, and gives final centroid locations. Explanation of specific lines of code can be found below.

```
In [21]:

for i in range(k):

    # select only data observations with cluster label == i

    ds = faith[np.where(labels==i)]

    # plot the data observations

    plt.plot(ds[:,0],ds[:,1],'o', markersize=7)

    # plot the centroids

    lines = plt.plot(centroids[i,0],centroids[i,1],'kx')

    # make the centroid x's bigger

    plt.setp(lines,ms=15.0)

    plt.setp(lines,mew=4.0)

plt.show()
```



Creating a visualization of the cluster model.

A quick breakdown of the code above:

1. All of the work done to group the data into 2 groups was done in the previous section of code where we used the command kmeans.fit(faith). This section of the code simply creates the plot that shows it.
2. The ds variable is simply the original data, but reformatted to include the new color labels based on the number of groups – the number of integers in k.
3. plt.plot calls the x-data, the y-data, the shape of the objects, and the size of the circles.
4. The rest of the code displays the final centroids of the k-means clustering process, and controls the size and thickness of the centroid markers.

And here we have it – a simple cluster model. This code can be adapted to include a different number of clusters, but for this problem it makes sense to include only two clusters. Now that we have these clusters that seem to be well defined, we can infer meaning from these two clusters. What do they stand for? The green cluster: consisting of mostly short eruptions with a brief waiting time between eruptions could be defined as 'weak or rapid-fire', while the blue cluster could be called 'power' eruptions.

Of note: this technique is not adaptable for all data sets – data scientist David