PW2-Data preparation :
A basic Machine Learning algorithms to Read/write files, deal with Missing, cleaning, and Impute data

Requirements for the work: the exercise 6 should be uploaded on the DVO repository before the deadline. You should create your colab file correctly named.

1 Learning outcomes PW2

In this lesson, we are faced with the data preparation before processing them with a ML pipeline. Data preparation is seen as an tremendous step in the ML pipeline, where data scientists spend much of their time cleaning up datasets and shaping them for better exploitation. In fact, a lot of data scientists argue that the initial steps of obtaining and cleaning data constitute 80% of the job. Your practical working will be splited into 3 parts as teh following:

1. Part I: Pandass Data toolkit to manage data from CSV format
2. Part II: Data checking
3. Part III: data repairing

Part I: Data management with Pandas

Pandas is used to manipulate, clean, and query data by looking at the Pandass data tool kit. Pandass was created by Wes McKinny in 2008 and is an open source project under a very permissive license.

Exercise 1: Pandss Series data structure

If you are familiar with Pandas and Series manipulation, you can skip this one and go to the second exercise. To start, you will create a new notebook using your local environment or colab environment. You have to name your notebook with PW2-[firstName_Lastname_MajorName]. Now you can answer the following questions

1. import Pandas and create a list of sports containing this list $['Football','HandBall','SnowSport']$ and store it in a variable **sports**;
2. cast your list as a series and display sports using the Class Pandass.Series (see documention on **class Pandass.Series(data=None, index=None, dtype=None, name=None, copy=None, fastpath=False)**)

3. what is the type of each element of your list?
4. Create a new list named **numeric** containing a short list of numbers and display it as a series.
5. add the value **None** to both **animals** and **numeric** and compare their corresponding display with Series. What did you remark?

```
numeric=[None, 15,100]
# cast list as a Series
pd.Series(numeric)
# create list of sports
sports = ['BasketBall', 'HandBall', None]
# cast list as a Series
pd.Series(sports)

```

6. We shall correct the incomplete value None of sports by changing the None by NaN (Numpy.nan)

```
import numpy as np
np.isnan(np.nan)
sports = ['BasketBall', 'HandBall', np.nan]
pd.Series(sports)

```

7. Let construct our variable sports from a dictionary as follows: {'BasketBall', 'HandBall', 'Snowsport', 'baseBall', 'Swimming'}. Cast the variable to a serie and store it a new variable **sIndex**. Display sIndex and call the function index on it. What did you remark?

Note: Series data structure

- Pandass.Series offers an object storage from a dictionary, the indexes become the keys rather than integers
- You need to use special functions to test the presence of a **not a number**, such as the numpy library **isnan() function**.

Exercise 2: Series Querying

Let we consider the variable sports from a dictionary.

```
import numpy as np
np.isnan(np.nan)
sports={'bask': 'BasketBall', 'hand': 'HandBall', 'snow': 'Snowsport', 'base': 'baseBall',
    'swim': 'Swimming'}
#cast the list on a serie
sIndex=pd.Series(sports)
#display the serie
display(sIndex)
sIndex.index

```

1. Display the second element of the variable sports and find the element who has 'swim'. (use $iloc[]$ to index location and $loc[]$ to label location).
2. Transform all items of your list sports in an uppercase characters (***see Series.str.upper()***)
3. write a simple loop to transform sports in an uppercase Characters
4. compare the runtime between both solutions and explain the runtime gap (***see %timeit which is an IPython magic function, which can be used to time a particular piece of code***.)
5. compare the runtime between the function np.mean() and a loop to calculate the mean value of your numeric variable.

```
#Solution 1 with vectorisation using np.mean()
numeric=[None, 15,100]
# cast list as a Series
num=pd.Series(numeric)
meanValue=np.mean(num)
print(meanValue)
#Solution 2 using a loop
meanValue = 0
sum=0
for item in num:
    ....#be carefull: you have to manage the NaN value
meanValue=sum/len(num)

```

Note: Vectorization and broadcasting operators

- Vectorization in Pandass allows to apply a numerical operation on all values of a serie like the mean, sum, etc.
- Broadcating in Pandass allows to apply an operation to each value on the serie.

Part II: data checking

Exercise 3: From CSV to DataFrame data structure

As a first step, we need python libraries allowing us to load our data as a dataframe. You have to download the csv file named ***Pearson.csv***. This file contains 15 columns separated with a ***',' character***.

1. Load the file ***Pearson.csv*** and store its content into a dataframe variable ***pearson_Data***

```
    #be carefull: you need the relative path to your file folder
    pearson_Data =pd.read_csv('input/Mydatasets/Pearsons.csv', delimiter= ',')
```

2. Display the shape of your dataframe and display the header to understand the meaning of each column.
3. Use the ***dataframe.columns*** to display the dictionary indexing your data

Exercise 4: Check missing values

Now we want to evaluate the number of missing data in the hole data. Display the total of missing values in each column of ***pearson_Data***

1. You can use ***dataframe.isnull()*** and ***verctorization*** to sum missing values by each column
2. Display the rows with missing values in the ***'name' column***, you should have 81 rows
3. Try removing rows with missing names. We call ***dropna()*** function which has many arguments to scale the level of a filtered data:
   - **axis=0 or 1** allows to filter the missing values according to the **row(axis=0)** or the **column(axis=1)**
   - **how=all** allows to eliminate all rows or all columns having mainly missing values
   - **inplace = True** allow to need the same dataframe without creating a new output
   - **thresh= integer value** allows to keep only rows (or columns) having a missing value rate more than thresh.

   test the function dropna() on your dataframe as:
   - ***dropna(how='all', inplace=True)*** and check if missing values are removed!
   - ***dropna(inplace=True, axis=0)*** and check if missing values are removed!

   What did you conclude?

Exercise 5: Ensure that values have the right format/type

If we take the "name" column, we find the first name and the last name separated with a blank character

1. check the type of each column using the function ***astype()***
2. Let we split the column 'name' into two columns where the first string is the first name and the second string is the last name. (***see str.split()***).
3. You can remark incorrect ***lastname*** values. Remove rows having lastname length up to 16 characters.

Take 5 mn to outline the main ideas you've retained from these exercises (Part I and II)

Part III: Data repairing with imputation

**Exrcise 6: Do it from scratch at home and upload it on Moodle

Now you have a new dataset "Olympics.csv" available on Moodle. You have to define your own strategy to clean this data and comment your notebook at each step.

Exercise 7: Imputation to handle missing data

In this exercise we will compare two solutions of data cleaning. The first one consists to drop missing columns values while the second is to fill missing values with interesting ones. In your notebook, open the file ***melb_data.csv***.

1. load your data as a dataframe and use a clear name as ***"initialData"***
2. observe first the quality of the data by using these functions: dataframe.columns, dataframe.info(), and dataframe.shape.
3. display the total missing values by columns (axis=0)
4. display total missing Values by rows (axis=1)
5. list the columns with missing values and store them in a variable ***colsWithMissing***. Use the function isnull() and try to write a simple function you can call it with other datasets with the following signature and core code as following:

```
    def  missing_colums(originDB):
        #  core code ....
         return colsWithMissing, reduced_original_data
    #Core code is :
    #use an iterator on  initialData.columns and a vectorization mode
    colsWithMissing = [col for col in origin_Data.columns
                                if origin_Data[col].isnull().any()]
    reduced_original_data = origin_Data.drop(colsWithMissing, axis=1)

```

   If those columns had relevant information your model loses access to it when the column is dropped. Another drawback to this solution is to miss to do the same droping on the test dataset where an error will occur.

6. display the rate of missing values by columns
7. do the same on the rows
8. remove rows whom the rate of missing values are $> 5\%$ from the origin data and store the result on a new dataframe variable named new_Data.
9. call your function missing_colums(originDB) on both original data and on new data obtained after removal rows. How do you explain the columns difference?
10. fill the missing values with the mean price, so you have to: 1) Display the statistical description of the column ***Price*** using the function describe(), 2) then to calculate the mean value and 3) to fill the missing value with the mean:

```
        new_Data.loc[:,'Price'].describe()
        new_Data['Price'].mean()
        new_Data.loc[:,'Price'].fillna(new_Data['Price'].mean(),inplace=True)

```

Exercise 8: Transformer and Pipeline with scikitlearn for Imputation processing

In the previous exercise, we have decided to estimate the missing values using the mean estimator and fill missing values with the same value. However the imputation is an estimation of the data with a defined strategy as the median, the mean, variance, etc. Scikitlearn API provides an simple class to take care of missing values: ***SimpleImputer***. To use this class, we have to

- create a SimpleImputer instance by specifying the strategy

```
        from sklearn.impute import SimpleImputer
        my_imputer = SimpleImputer(strategy='mean')

```

- since the mean could be calculated only on numerical attributes, you need to create a copy of the data without the text attributes. You call this copy as a ***origin_Data_Num***
- Then you have to fit the mean on your ***origin_Data_Num***

```
        from sklearn.impute import SimpleImputer
        my_imputer = SimpleImputer(strategy='mean')
        Origin_Data_num=origin_Data.drop(...)
        my_imputer.fit(Origin_Data_num)


```

  the fit() function stores the result into its statistic instance variable. At any moment you can reuse this variable to repair new missing values. You can display this variable: imputer.statistics_

- finaly You use the imputer to transform the numerical data:

```
        from sklearn.impute import SimpleImputer
        my_imputer = SimpleImputer(strategy='mean')
        Origin_Data_num=origin_Data.drop(...)
        my_imputer.fit(Origin_Data_num)
        my_imputer.statistics_
        my_imputer.transform(Origin_Data_num)

```

Be careful, the output of the trsaform function is a numpy array. You have to convert it to a dataframe:

```
        from sklearn.impute import SimpleImputer
        my_imputer = SimpleImputer(strategy='mean')
        Origin_Data_num=origin_Data.drop(...)
        my_imputer.fit(Origin_Data_num)
        my_imputer.statistics_
        Transform_Data_num = my_imputer.transform(Origin_Data_num)
        df_Transform_Data_num=pd.DataFrame(Transform_Data_num, Origin_Data_num.columns,
index=Origin_Data_num.index)

```

Exercise 9: to learn more

You can create a pipeline to automate the cleaning process using the pipe and imputer. Reshape your code to 1) create a pipeline to handle numerical data and an other pipeline to handle categorical data.