Kingdom of Morocco
Al Akhawayn University in Ifrane
School of Science and Engineering

Lab Manual

Machine Learning and Data Mining

CSC3347

Amine Abouaomar, Ph.D.
a.abouaomar@aui.ma

(Version 1.0)
September 2023

Lab 1: Data Preprocessing and Visualization

- **Objective**

Learn how to preprocess and visualize datasets using Python libraries, namely, Pandas and Matplotlib.

Tools needed,

- Python3
- Pip installer
- You can use whatever installation mode (bare metal, Anaconda, containers, ...etc.)

Tools setup,

1- Install the adequate libraries.
   a. Pandas

- Pandas is a Python library designed for data manipulation and analysis.
- It offers a wide range of functions for tasks such as data cleaning, exploration, analysis, and manipulation.
- The name "Pandas" is derived from "Panel Data" and "Python Data Analysis."

```
$: pip install pandas
```

To test that the library is successfully installed, try to import in python,

```
$: python

Python 3.11.4 ...

>>> import pandas
```

If no error message was displayed nor exception was arised, then, pandas library is installed.

Display the version of pandas,

```
>>> pandas.__version__
```

You can use aliases to import libraries, to make your code lighter. Aliases are created using the keywork as,

```
>>> import pandas as pd
```

To access the __version__ attribute, use,

```
>>> pd.__version__
```

Now, the Pandas package can be referred to as **pd** instead of **pandas**.

Note that all the programs we will be designing and coding in these labs should be written as programs, instead of being executed line by line in the terminal.

b. Matplotlib

- Matplotlib is a Python library used for creating static, animated, and interactive visualizations.
- It provides a wide range of functions and tools for generating high-quality 2D and 3D plots and charts.
- Matplotlib is often used for data visualization, scientific plotting, and creating graphical representations of data.

We move forward with matplotlib the same way we did with pandas.

```
$: pip install matplotlib
```

If the library is successfully installed, there is no further task to perform until data visualization part.

2- Pandas series

a. About series

Pandas Series is a one-dimensional labeled array. It is a fundamental data structure provided by the Pandas library, which is commonly used for data manipulation and analysis.

```
import pandas as pd
arr = [13.4, 13.8, 12.9, 13.1, 14]
serArr = pd.Series(arr)
print(serArr)
```

You can see Pandas Series as a column in a table. A one-dimensional array used to store data of any type.

With series, data is labeled through its index number, starting from 0 to the size-1. Labels are used to access specified values.

b. Labels

Another way to index data is through labels, where, instead of using numerical indexing, we use keywords. These keywords are also referred to as labels.

With the **index** argument, you can give names to labels.

```
serArr = pd.Series(arr, index = ["Jan", "Feb", "Mar", "Apr", "May"])
print(serArr["Jan"])
```

In this example, we can access the data by referring to the label ("Jan" in the example).

c. Key/value objects

Key/value objects, often referred to as key-value pairs or simply key-value objects, are a fundamental data structure used in computer science and programming. They consist of two parts,

1- **Key**: A unique label that is used to reference the associated value. Keys can be strings, numbers, or other hashable data types and must be unique within a given collection of key-value pairs.
2- **Value**: The data associated with a specific key. Values can be of any data type, including numbers, strings, lists, dictionaries, objects, or even other key-value pairs.

We can use a key/value object, as a dictionary, in creating a Series.

```
# initial version of the code
arr = [13.4, 13.8, 12.9, 13.1, 14.0]
serArr = pd.Series(arr, index = ["Jan", "Feb", "Mar", "Apr", "May"])


# with a dictionary
arr_d = {"Jan": 13.4, "Feb": 13.8, "Mar": 12.9, "Apr": 13.1, "May":14.0}
serArr_d = pd.Series(arr_d)
```

To select only some of the items in the dictionary, use the index argument to specify the items you want to include in the Series.

```
# with a dictionary
arr_d = {"Jan": 13.4, "Feb": 13.8, "Mar": 12.9, "Apr": 13.1, "May":14.0}
serArr_d = pd.Series(arr_d, index=["Jan", "Feb"])
```

d. Data frames (Dataframes)

A DataFrame (DF) is a two-dimensional, tabular data structure commonly used in data analysis, data manipulation, and data visualization. It is a fundamental concept in data science, that can be thought of as tables or spreadsheets, similar to what you might encounter in a database or Excel.

Here is some key characteristics of tabular data,

- **Tabular structure**: DFs consist of rows and columns, similar to a table in a relational database or an Excel spreadsheet. Rows represent a record or observation, while each column represents a variable or attribute.
- **Labeled axes**: DFs have labeled rows and columns. Row labels are often referred to as indexes, while column labels are the variable names.
- **Heterogeneous data types**: Columns in a DFs can contain data of different types, including numbers, strings, dates, and more. This allows working with diverse datasets where different variables may have different data types.
- **Data alignment**: DFs automatically align data based on labels, making it easy to manipulate and perform different operations on corresponding elements across different columns.
- **Missing data handling**: DFs provide mechanisms for handling missing data, such as NaN (Not a Number) values. This is essential for working with real-world datasets that may have incomplete information.
- **Flexibility**: DFs support various data manipulation operations, including filtering, grouping, aggregation, merging, and reshaping. This makes them a powerful tool for data analysis and transformation.

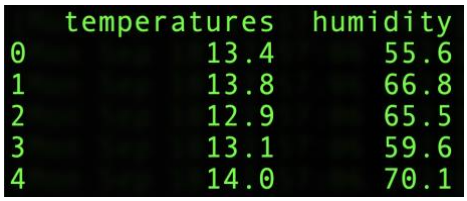
We can create data from different series, such as shown in the following example,

```
import pandas as pd


temp = [13.4, 13.8, 12.9, 13.1, 14.0]
humid = [55.6, 66.8, 65.5, 59.6, 70.1]
data = {
   "temperatures" : temp,
   "humidity": humid
   }
```

```
df = pd.DataFrame(data)
print(df)
```

This will produce the following result, which shows a dataframe as rows and columns.



	temperatures	humidity
0	13.4	55.6
1	13.8	66.8
2	12.9	65.5
3	13.1	59.6
4	14.0	70.1

Figure 1 Results of the dataframe example.

We can return a specific row or rows using the loc attribute of the df object.

```
df = pd.DataFrame(data)
# access the data in the index 0 of the dataframe
print(df.loc[0])
# access the data is the index 0 and 2 of the dataframe
print(df.loc[[0, 2]])
```

We can also use indexed columns through specifying the columns using the attribute index of the DataFrame constructor.

```
df = pd.DataFrame(data, index = ["Jan", "Feb", "Mar", "Apr", "May"])
```

We can also use this indexing way to locate data.

```
df = pd.DataFrame(data)
# access the data in the index 0 of the dataframe
print(df.loc[0])
# access the data is the index 0 and 2 of the dataframe
print(df.loc[["Jan", "Feb"]])
```

3- Load a data from a file.

To load data from a file, we use the method read_csv.

```
df = pd.read_csv('data.csv')
```

read_csv takes the file name as parameters, in this case data will be loaded from data.csv.

data.csv of course should meet some requirements in terms of the format.

a. CSV files

Comma-separated values (CSV) is a text file format that uses commas to separate values. A CSV file stores tabular data either numbers or text in plain text, where each line of the file typically represents one data record.

An example of these files is given below. The content of the CSV file contains information as it is represented in the following table.

Year	Make	Model	Description	Price
1997	Ford	E350	ac, abs, moon	3000.00
1999	Chevy	Venture "Extended Edition"		4900.00
1999	Chevy	Venture "Extended Edition, Very Large"		5000.00
1996	Jeep	Grand Cherokee	MUST SELL! air, moon roof, loaded	4799.00

The first line in a CSV file is referring to the columns; in Pandas jargon, labels. The other lines contain data, and data can be inline as in lines 2 to 4, or multiple lines, as in line 5.

```
1 Year,Make,Model,Description,Price
2 1997,Ford,E350,"ac, abs, moon",3000.00
3 1999,Chevy,"Venture ""Extended Edition""","",4900.00
4 1999,Chevy,"Venture ""Extended Edition, Very Large""","",5000.00
5 1996,Jeep,Grand Cherokee,"MUST SELL!
  air, moon roof, loaded",4799.00
```

b. Viewing data

```
df = pd.read_csv('data.csv')
# to print the plain data from the loaded dataset, we use the to_string() method
```

```
print(df.to_string())
```

To specify the maximum number of lines to be loaded from the file, use the *pd.options.display.max_rows = MAX_NUMBER*.

The most used method for getting a quick overview of the DataFrame, is the head() method. This method has an optional parameter to specify the number of lines to be returned.

```
df = pd.read_csv('data.csv')


print(df.head())       # by default, it returns 5 rows.
print(df.head(10))     # will return 10 lines
```

Another method, is tail(), that prints the last 5 lines by default, or the number of lines passed as argument.

```


print(df.tail())       # by default, it returns 5 rows.
print(df.tail(10))     # will return 10 lines]
```

Info method is used to gives us more information about the data set.

```
print(df.info())
```

Task 1: Analyze the result of the info method.

4- Cleaning data

Data cleaning means fixing bad data in your data set.

Bad data could be:

- Empty cells
- Data in wrong format
- Wrong data
- Duplicates

We will tackle each of these cases in the following subsections.

i. Empty cells

One way to deal with empty cells is to remove completely the rows that contain empty cells. It is usually fine to do so, since data sets can be very big, and removing a few rows will not impact on the result.

```
df = pd.read_csv('data.csv')
# dropna method removes all the enteries in
# the dataset that contains NaN values
new_df = df.dropna()
print(new_df.to_string())
```

Note: By default, the dropna() method returns a new DataFrame, so that no changes will affect the original data frame.

If you want to change the original DataFrame, use the inplace = True argument:

```
df.dropna(inplace = True)
```

Another way is to change the NaN values into a default value.

Note: The dropna(inplace = True) will NOT return a new DataFrame, instead, it will remove all rows containing NULL values from the original DataFrame.

An alternative approach to handling empty cells is to substitute them with a new value. This method eliminates the need to remove entire rows due to a few vacant cells. The fillna() function enables us to fill empty cells with a specified value.

```
df.fillna(130, inplace = True)
```

This example fills all empty cells across the entire DataFrame. If you want to replace empty values in a specific column only, mention the column name of the DataFrame.

```
df["Jan"].fillna(130, inplace = True)
```

One frequent method to fill empty cells is by computing the mean, median, or mode of the column. In Pandas, the mean(), median(), and mode() functions are used to determine these values for a given column.

```
df = pd.read_csv('data.csv')
x = df["Jan"].mean()
df["Jan"].fillna(x, inplace = True)
```

Note: The "mean" or "average" of a set of numbers is calculated by adding up all of the numbers and then dividing by the count of those numbers.

We can use the median.

```
x = df["Jan"].median()
df["Jan"].fillna(x, inplace = True)
```

Note: Median = the value in the middle, after you have sorted all values ascending.

We can use the mode as well

```
x = df["Jan"].mode()[0]
df["Jan"].fillna(x, inplace = True)
```

Note: Mode = the value that appears most frequently.

Task 2: Deal with empty cells using one or more of the discussed methods.

ii. Data in wrong format

Cells containing improperly formatted data can hinder or even prevent data analysis. To address this, you can either eliminate the problematic rows or standardize the format of all cells within the affected columns.

In the provided data set, there are many rows with different date format. We need to give it all the same format.

We will convert the bad values to a valid date format.

```
df['Sale Date'] = pd.to_datetime(df['Sale Date'])
```

From the outcome, the dates in rows with incorrect formatting were corrected, but the blank dates were assigned a NaT (Not a Time) value, essentially indicating a missing value. One approach to handle such missing values is to delete the respective row entirely.

```
df.dropna(subset=['Date'], inplace = True)
```

After the conversion in the example, we obtained a NaT value. This can be treated as a NULL value. To eliminate rows with such values, the dropna() method can be utilized.

Task 3: Find and correct the wrong format of the data in the provided data.

iii. Wrong data

"Wrong data" does not have to be "empty cells" or "wrong format", it can just be wrong, like if someone registered "199" instead of "1.99". Sometimes we can spot wrong data by looking at the data set, because you have an expectation of what it should be.

In the provided data set, you can see that in row 3, 16, and 23, the selling prices is negative, but for all the other rows the price is positive. It doesn't have to be wrong, but taking in consideration that this is sales data, we conclude with the fact that a car cannot be sold with a negative price.

A method to correct erroneous values is to substitute them with alternative data. In the given example, it appears that the negative values are due to a typographical error. For smaller data sets, it might be feasible to manually amend each incorrect value, but this isn't practical for vast data sets.

For extensive datasets, we can establish specific criteria, like setting acceptable value ranges, and then substitute any data that falls outside these ranges.

In our case, to address this, simply multiply these values by -1.

```
for x in df.index:
    if df.loc[x, "Sale Price"] < 0:
        df.loc[x, "Sale Price"] *= -1
```

An alternative approach to dealing with incorrect data is to simply delete the rows containing such data. By doing so, you eliminate the need to determine suitable replacement values, and there's a high likelihood that these rows aren't essential for your analysis.

```
for x in df.index:
    if df.loc[x, " Sale Price"] < 0:
        df.drop(x, inplace = True)
```

Task 4: Deal with the wrong values of data.

iv. Duplicates

Duplicate rows refer to entries that appear multiple times in a dataset.

You can discover duplicates, either by eyes, or using method/functions from Pandas library.

To identify duplicate entries, the duplicated() method can be employed. The duplicated() method provides Boolean values for every row, indicating whether it's a duplicate or not.

```
print(df.duplicated())
```

Expect a result that looks like this,

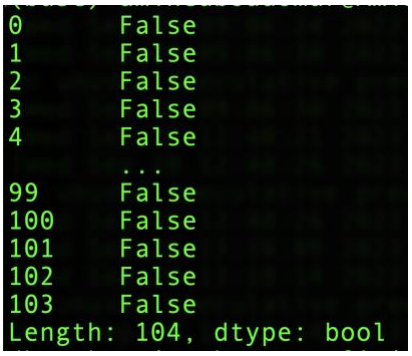


Figure 2 Output of the duplicated method call.

To remove duplicates, use the drop_duplicates() method.

```
df.drop_duplicates(inplace = True)
```

Keep in mind: Using (inplace = True) ensures that the method won't produce a new DataFrame. Instead, it will remove all duplicates directly from the original DataFrame.

Task 5: Remove duplicated data entries if this is relevant to your dataset.

5- Plotting

Pandas employs the plot() method to generate diagrams. To display the diagram on the screen, we can utilize PyPlot, which is a submodule of the Matplotlib library.

```
import pandas as pd
import matplotlib.pyplot as plt


df = pd.read_csv('dirty_data.csv')
df.plot()
plt.show()
```

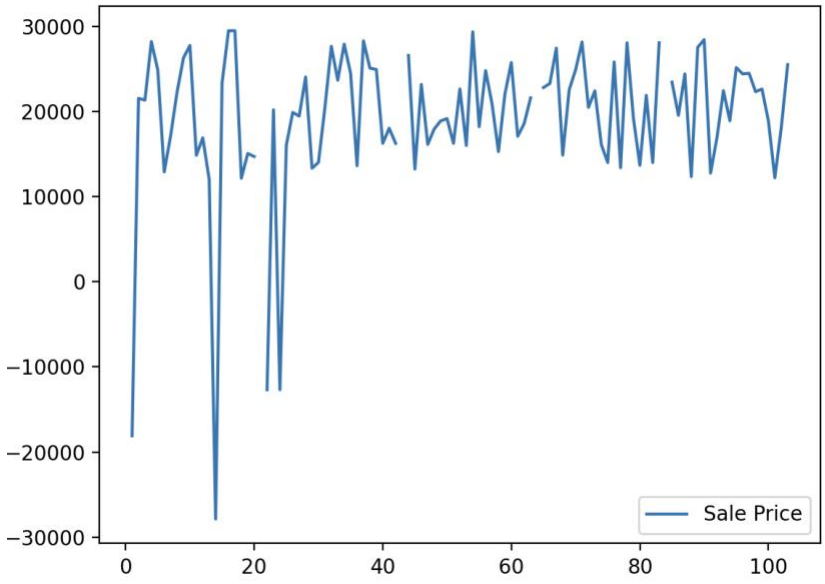


Figure 3 Plotting of the sales price.

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('dirty_data.csv')
df.plot(kind = 'scatter', x = 'Sale Date', y = 'Sale Price')
plt.show()
```

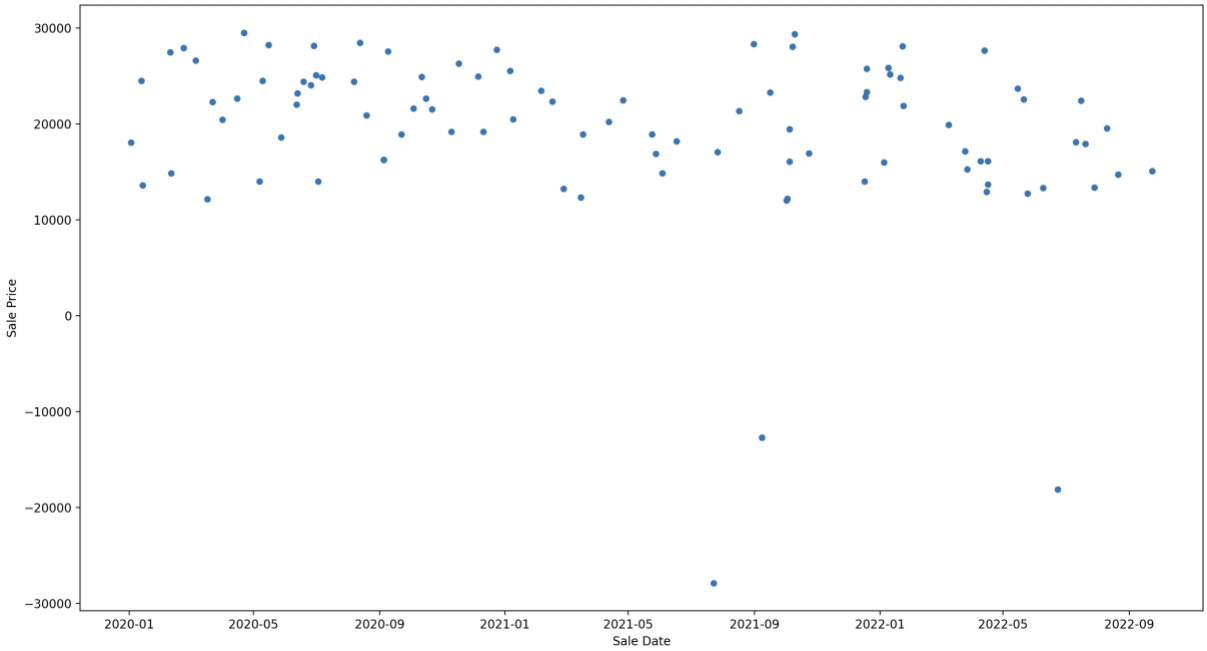


Figure 4 Scatter plotting.

To create a histogram using pandas, specify the type of plot using the kind argument, (kind = 'hist')

Unlike a scatter plot, a histogram is used to represent the distribution of a single column's data, showing the frequency of occurrences in different intervals. For instance, if you want to visualize how often workouts fall within certain duration ranges. We will use a histogram. In the following example, we'll utilize the "Sale Price" column to generate the histogram.

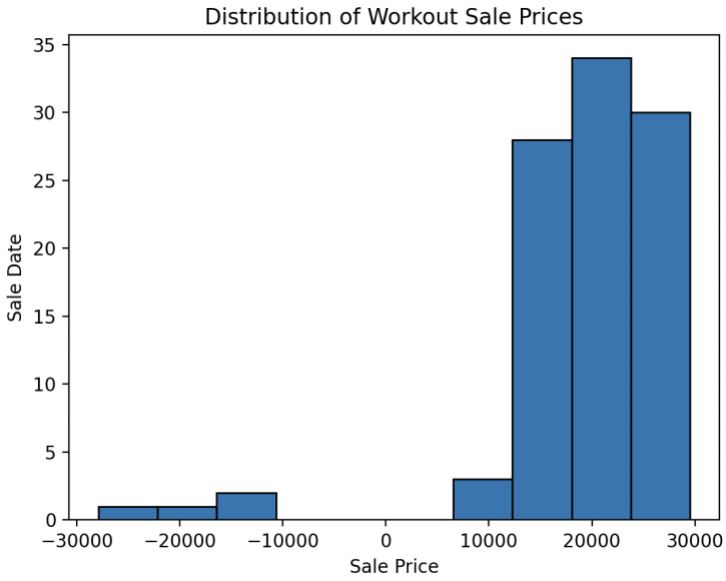


Figure 5 Plotting histogram example.