0201-solution-notebook

March 19, 2024

1 0201 - Advanced EDA With Python - Solution Notebook

- Written by Alexandre Gazagnes
- Last update: 2024-02-01

1.1 About

1.1.1 Using Jupyter

You have 2 options: - Locally:

```
- **Install Anaconda https://www.anaconda.com/ or Jupyter https://jupyter.org/install on your m

- Use Anaconda or Jupyter installed on the Unilasalle PC (**Warning **: some packages may be m
```

- Online:
  - **Use Google Colab https://colab.research.google.com/** (you have to be connected to your google account)
  - **Open this notebook on Google colab URL**
    - Badge
  - Use Jupyter online https://jupyter.org/try-jupyter (**Warning** : External packages cannot be installed)

1.1.2 Material

All the material for this course could be found here. - https://github.com/AlexandreGazagnes/Unilassalle-Public-Ressources/tree/main/4a-data-analysis

1.1.3 Context

You're an export project manager for a major food manufacturer.

You are in charge of poultry departement.

You have been asked to identify countries within the company's database or using external datasets in order to target them.

1.1.4 Data

After a quick look on the internet, you find a very interesting dataset on the FAO website. It contains a list of countries with various indicators. You decide to use this dataset to identify segments of countries.

Find the data :

- **You can use a preprocessed version of the dataset here.** (Best option)
- You can also download the "raw" dataset here. (**Warning** : You will have to preprocess the data before playing this notebook)

1.1.5 Mission

Your objective is to :

- Take a quick tour of the data to understand the data set
- Clean up the dataset if necessary
- Try to have a deep understanding of our market, using, PCA or MCA.
- Perform clustering with Kmeans and Agglomerative Clustering, focusing on countries with large potential markets: populous countries, wealthy countries and/or countries with high import levels
- You need to be able to understand and explain the clusters you've created.

1.2 Preliminaries

1.2.1 System

These commands will display the system information:

Uncomment theses lines if needed.

[ ]:
```
# pwd
```

[ ]:
```
# cd ..
```

[ ]:
```
# ls
```

[ ]:
```
# cd ..
```

[ ]:
```
# ls
```

These commands will install the required packages:

Please note that if you are using google colab, all you need is already installed

[ ]:
```
# !pip install pandas matplotlib seaborn plotly scikit-learn
```

This command will download the dataset:

Please note that we will download the dataset later, in this notebook

[ ]:
```
# !wget https://gist.githubusercontent.com/AlexandreGazagnes/
↪28a8da40ffa339b96b02f3e3cd79792d/raw/
↪4849eba0d69f43472a7637e1b62e56fd7eb09c7e/chicken.csv
```

1.2.2 Import

Import data libraries:

[ ]:
```
import pandas as pd  # DataFrame
import numpy as np  # Matrix and advanced maths operations
```

Import Graphical libraries:

[ ]:
```
import matplotlib.pyplot as plt  # Visualisation
import seaborn as sns  # Visualisation

# import plotly.express as px       # Visualisation (not used here)
```

Import Machine Learning libraries:

[ ]:
```
from sklearn.preprocessing import StandardScaler  # Our Standardizer Obj.
from sklearn.decomposition import PCA  # Our PCA Obj.

# for Later :
# from sklearn.cluster import KMeans                          # Clustering
# from sklearn.cluster import AgglomerativeClustering         # Clustering
# from sklearn.metrics import silhouette_score                # Clustering
# from sklearn.metrics import davies_bouldin_score            # Clustering
# from sklearn.datasets import load_iris                      # Toy Dataset
# from scipy.cluster.hierarchy import dendrogram, linkage     # Clustering
```

:warning:**These imports must be done, it is not possible to use this notebook without pandas, matplotlib etc.**

1.2.3 Data

1st option : Download the dataset from the web

[ ]:
```
url = "https://gist.githubusercontent.com/AlexandreGazagnes/
↪28a8da40ffa339b96b02f3e3cd79792d/raw/
↪4849eba0d69f43472a7637e1b62e56fd7eb09c7e/chicken.csv"
df = pd.read_csv(url)
df.head()
```

2nd Option : Read data from a file

[ ]:
```
# # or

# fn = "my/awsome/respository/my_awsome_file.csv"
# fn = "./chicken.csv"
```

```
# df = pd.read_csv(fn)
# df.head()
```

1.3 Data Exploration

1.3.1 Display

Display the first rows of the dataset:

[ ]:
```
# Head

df.head(20)
```

Display the last rows of the dataset:

[ ]:
```
# Tail

df.tail()
```

Display a sample of the dataset:

[ ]:
```
# Sample

df.sample(20)
```

[ ]:
```
# Sample with just 10% of the dataset

df.sample(frac=0.10)
```

1.3.2 Structure

What is the shape of the dataset?

[ ]:
```
# Structure

df.shape
```

What data types are present in the dataset?

[ ]:
```
# Dtypes

df.dtypes
```

:warning: **Please note that we have here main python dtypes** Data types : - int : *Integer* : 1,2,12332, 1_000_000 - float : *Float* : 1.243453, 198776.8789, 1.9776 - object : In this example object stands for *String* : “Paris”, “Rouen”, “Lea”

Get all the columns names:

[ ]:
```
# Info

```

```
df.info()
```

Count the number of columns with specific data types:

[ ]:
```
# Value counts on dtypes

df.dtypes.value_counts()
```

Select only string columns:

[ ]:
```
# Select dtypes str

df.select_dtypes(include="object").head()
```

Select only numerical columns:

[ ]:
```
# Select dtypes float

df.select_dtypes(include="float").head()
```

Count number of unique values :

[ ]:
```
# Number unique values for int columns

df.select_dtypes(include=int).nunique()
```

[ ]:
```
# Number unique values for float columns

df.select_dtypes(include=float).nunique()
```

[ ]:
```
# Number unique values for object columns

df.select_dtypes(include="object").nunique()
```

1.3.3 NaN

How many NaN are present in the dataset?

:warning: **Please note that `NaN` stands for Not A Number aka `Missing Value`***

[ ]:
```
# isna ?
df.isna().head()
```

[ ]:
```
# Sum of isna

df.isna().sum()
```

Do we have some Missing values here ?

1.3.4 Data Inspection

Have a look to a numercial summary of the dataset:

[ ]:
```
# Describe ?
df.describe()
```

Ok, hard to read… Let's try something better :

[ ]:
```
# Better ?
df.describe().round(0)
```

Compute the correlation matrix:

[ ]:
```
# creating tmp variable

corr = df.select_dtypes(include="number").corr()
corr.round(4)
```

Ok … Not so easy to read !

Try a first visualization of the correlation matrix:

[ ]:
```
# Building heatmap

sns.heatmap(corr, annot=True)
```

So far, so good, we have to improve this visualisation!

[ ]:
```
# Better heatmap ?
sns.heatmap(corr, annot=True, cmap="coolwarm", fmt=".2f", vmin=-1, vmax=1)
```

Great ! … Do we have something better ? (better stands for *more readable*)

Find the best visualization for the correlation matrix:

[ ]:
```
# Best heatmap ?
mask = np.triu(corr)
sns.heatmap(corr, annot=True, cmap="coolwarm", fmt=".2f", vmin=-1, vmax=1,␣
 ↪mask=mask)
```

We are gona use this correlation matrix a lot. Maybe it is a good idea to create a function !

Write your first function :

[ ]:
```
def add(a, b):

    return a + b
```

Try it :

[ ]:
```
add(1, 1)
```

Just add a triple quote `""" Some usefull comment """` after the first line :

[ ]:
```
def add(a, b):
    """This function make the addition of a and b and return the result"""

    return a + b
```

Write a function to display the correlation matrix:

[ ]:
```
# With a function


def make_corr_heatmap(df):
    """Just compute and plot the correlation matrix with a fancy heat map"""

    corr = df.select_dtypes(include="number").corr()
    mask = np.triu(corr)
    sns.heatmap(
        corr, annot=True, cmap="coolwarm", fmt=".2f", vmin=-1, vmax=1, mask=mask
    )
```

Use it :

[ ]:
```
make_corr_heatmap(df)
```

:clap: Nice ! We will use this visualization a lot in this feature...

1.3.5 Visualization

Use a Boxplot to visualize the distribution of the numerical columns:

[ ]:
```
# Box plot
sns.boxplot(data=df.population)
```

We have a very unbalanced distribution with 2 outliers : China and India

Please note that China do not have 1.4 million citizens but 1.4 BILLION. So we need to fix this (later).

Just as fun fact, we can use the log trasnformation to *fix* this distribution.

Try to apply log transformation to the numerical columns:

[ ]:
```
ser = np.log(df.population)
ser
```

Display the box plot of our new feature :

[ ]:
```
sns.boxplot(ser)
```

:clap: Great !

Let's try an other implementation (same result) :

[ ]:
```
tmp = np.log1p(df.population)  # Here np.log1p(my_feature) == np.log(1 +␣
↪my_feature)
sns.boxplot(data=tmp)
```

Why `log(my_feature + 1 )` ?

Imagine all the people … like John Lenon… or better ! Imagine NO PEOPLE, 0 for the population of one country…

What is the output of `np.log(0)` ?

Plot all numerical columns:

[ ]:
```
sns.boxplot(data=df.select_dtypes(include="number"))
```

Plot each numerical column:

[ ]:
```
for col in df.select_dtypes(include="number").columns:
    plt.figure()
    sns.boxplot(data=df[col])
```

Make a pairplot of the numerical columns:

This visualization can be slow with large datasets.

Use `VIZ = True / False` to enable / disable the visualization.

[ ]:
```
VIZ = True  # Enable this with True
if VIZ:
    sns.pairplot(df.select_dtypes(exclude="object"), corner=True)
```

1.4 Data Cleaning

1.4.1 Population

Have a look to small countries :

[ ]:
```
df.population.describe().round(0)
```

Update the population with the good number :

[ ]:
```
df.population = df.population.astype(int) * 1_000
df.population.describe().round(0)
```

Sort the dataset by population :

[ ]:
```
df.sort_values("population", ascending=False).head()
```

[ ]:
```
df.sort_values("population", ascending=True).head()
```

Remember the shape of the dataset

[ ]:
```
df.shape
```

Select only “large” countries +1M :

[ ]:
```
threshold = 1000000
```

Here `1000000` is not so easy to read.

With python, we can use the _ special character in order to separate thousands :

[ ]:
```
threshold = 1_000_000
```

Trying to find rows with pop > threshold :

[ ]:
```
df.population > threshold
```

Creating a tmp (temporary) variable.

[ ]:
```
indexor = df.population > threshold
indexor
```

Use this `bool` vector (True / False) as a selector :

[ ]:
```
df = df.loc[indexor]
df
```

Of course, you can write all this code in a single line :

[ ]:
```
df = df.loc[df.population > 1_000_000]
df
```

What about our new DataFrame?

[ ]:
```
df.sort_values("population", ascending=True).head()
```

Ok why not, but let’s try a new threshold.

Select only “large” countries +5M :

[ ]:
```
threshold = 5_000_000
df = df.loc[df.population > threshold]
df
```

[ ]:
```
df.sort_values("population", ascending=True).head()
```

What about our correlation matrix :

[ ]:
```
make_corr_heatmap(df)
```

1.4.2 Columns

Select only relevant columns:

[ ]:
```
cols = [
    "code_zone",
```

```
    "zone",
    "dispo_int",  # WHY NOT
    "import",
    # "dispo_prot",
    "dispo_alim",
    "export",
    # "residus",
    # "var_stock",
    # "prod",
    # "nourriture",
    "population",
]

cols
```

Make the selection :

[ ]:
```
df = df.loc[:, cols]
df
```

[ ]:
```
make_corr_heatmap(df)
```

1.5 Feature engineering

Have a look to our dataset:

[ ]:
```
df
```

1.5.1 Dependency

Create a new column with some kind of “depedency” :

[ ]:
```
df["dependency"] = df["import"] / df["dispo_int"]
df
```

Sort the dataframe by dependency :

[ ]:
```
df.sort_values("dependency", ascending=False).head()
```

[ ]:
```
df.sort_values("dependency", ascending=True).head()
```

If we have any country with a `0` value for “‘dispo_int““ ?

So we need to clean `0` values for `dispo_int`.

Drop columns with infini values:

[ ]:
```
df = df.loc[df.dispo_int > 0]
df.sort_values("dispo_int", ascending=True).head()
```

Drop useless columns if needed :

df = df.drop(columns=[“code_zone”], errors=“ignore”) df

1.5.2 Delta

Compute diffrence between columns Import and Export :

[ ]:
```
# Compute Import - Export
# Create new column name delta

df["delta"] = df["import"] - df["export"]
df
```

Display the correlation matrix :

[ ]:
```
make_corr_heatmap(df)
```

Export is no more needed :

[ ]:
```
df.drop(columns="export", inplace=True, errors="ignore")
```

Last print of our df :

[ ]:
```
# Last print of our df

df
```

1.5.3 Scale

We need to standardize our dataset, ie transform it to have 0.0 as mean and 1.0 as std

[ ]:
```
df
```

Select only numerical columns:

:warning:**Using `X` here is a very strong convention applied by any data analyst, scientist etc.**

[ ]:
```
X = df.select_dtypes(include="number")
X
```

Init a `scaler` object from the `StandardScaler` class of scikit-learn :

[ ]:
```
scaler = StandardScaler()
scaler
```

Fit this object, ie *precompute* the transformations

[ ]:
```
scaler.fit(X)
```

Finally, do transform our dataset:

[ ]:
```
X_scaled = scaler.transform(X)
X_scaled
```

Please not that we can perform fit and transform in the same operation

[ ]:
```
X_scaled = scaler.fit_transform(X)
X_scaled
```

As usual... not so easy to read.

The have a `np.ndarray` oject, and the output is pretty uggly.

We can use the `type` function of python to have an idea of the data type of `X_scaled`

[ ]:
```
type(X_scaled)
```

Rebuild a DataFrame with the scaled data:

[ ]:
```
X_scaled = pd.DataFrame(X_scaled, columns=X.columns)
X_scaled.head()
```

Check that data were scaled:

[ ]:
```
X_scaled.describe().round(1)
```

Of course you can compute this transformation manually:

[ ]:
```
X_scaled = (X - X.mean()) / X.std()
X_scaled.head()
```

Ok, this implementation is better !

But we have used a `StandardScaler` for educational purpose : regarding OOP, regarding how to manage a scikit-learn object etc.

[ ]:
```
X_scaled.describe().round(2)
```

1.6 Principal Component Analysis

1.6.1 Init and fit

Initialize a PCA :

[ ]:
```
pca = PCA(n_components=6)
pca
```

Fit :

[ ]:
```
pca.fit(X_scaled)
```

Here is our new dataset :

[ ]:
```
X_proj = pca.transform(X_scaled)
X_proj
```

Here, `X_proj` is a strong convention

Same problem than with X_scaled : Not so easy to read.

Same problem ? Same solution!

Use pandas to create a DataFrame :

[ ]:
```
# About the columns names

cols = [
    "PC1",
    "PC2",
    "PC3",
    "PC4",
    "PC5",
    "PC6",
]
```

[ ]:
```
# Out new X_proj :

X_proj = pd.DataFrame(X_proj, columns=cols)
X_proj
```

Please note that a better implementation of our columns list is possible :

[ ]:
```
cols = [f"PC{i}" for i in range(1, pca.n_components_ + 1)]
cols
```

1.6.2 Analyse the components

Our components :

[ ]:
```
pcs = pca.components_
pcs
```

Using a data Frame :

[ ]:
```
components = pd.DataFrame(pcs, columns=X.columns, index=cols)
components
```

Recompute the first value :

[ ]:
```
X_proj
```

[ ]:
```
value = X_proj.iloc[0, 0]
value
```

1st line of X_scaled

[ ]:
```
X_scaled.head(1)
```

Compute our value :

[ ]:
```
(
    (-0.3724 * 0.66)
    + (-0.4424 * 0.11)
    + (-1.090 * 0.34)
    + (-0.1587 * 0.46)
    + (0.4643 * -0.1)
    + (0.1057 * -0.46)
)
```

`0.66`, `0.11` ... come from our components :

[ ]:
```
components
```

Insead of using an uggly sum, we can use just one line of code :

[ ]:
```
sum([i * j for i, j in zip(pcs[0], X_scaled.iloc[0])])
```

Just transpose this dataframe (Swap columns and rows for better readability) :

[ ]:
```
components = components.T
components
```

Add a Heatmap :

[ ]:
```
sns.heatmap(components, cmap="coolwarm", vmax=1, vmin=-1, annot=True, fmt=".2f")
```

1.6.3 Plot explained variance

The explained variance ratio is pre-computed by scikit-learn :

[ ]:
```
pca.explained_variance_ratio_
```

We can plot it :

[ ]:
```
sns.lineplot(y=pca.explained_variance_ratio_, x=components.columns, marker="o")
```

A better feature here is the cumulative variance :

[ ]:
```
cum_var = pca.explained_variance_ratio_.cumsum()
cum_var
```

We can plot it :

[ ]:
```
x = components.columns.tolist()
y = cum_var.tolist()
sns.lineplot(y=y, x=x, marker="o")
```

Just add `PC0` with a fake `0` value to improve our visualization :

[ ]:
```
x = ["PC0"] + components.columns.tolist()
y = [0] + cum_var.tolist()
sns.lineplot(y=y, x=x, marker="o")
```

1.6.4 Correlation graph

[ ]:
```
def correlation_graph(
    X_scaled,
    pca,
    dim: list = [0, 1],
):
    """Display the correlation graph for a PCA

    Positional arguments :
        - X_scaled : DataFrame | np.array : the scaled dataset
        - pca : PCA : a PCA instance already fitted

    Optional arguments :
        - dim : list or tuple : values x,y for the dimensions (PC) to plot.
            default :  [0,1] stands for F1, F2"""

    # Extract x et y
    x, y = dim

    # Features
    features = X_scaled.columns

    # Fig size (inches)
    fig, ax = plt.subplots(figsize=(10, 9))

    # For each component :
    for i in range(0, pca.components_.shape[1]):
        # Les flèches
        ax.arrow(
            0,
            0,
            pca.components_[x, i],
            pca.components_[y, i],
            head_width=0.07,
            head_length=0.07,
            width=0.02,
        )

        # Labels
        plt.text(
            pca.components_[x, i] + 0.05,
            pca.components_[y, i] + 0.05,
```

```
            features[i],
        )

    # Display horizontal and vertical lines
    plt.plot([-1, 1], [0, 0], color="grey", ls="--")
    plt.plot([0, 0], [-1, 1], color="grey", ls="--")

    # Axes names with % of inertia
    plt.xlabel(
        "F{} ({}%)".format(x + 1, round(100 * pca.explained_variance_ratio_[x],␣
↪1))
    )
    plt.ylabel(
        "F{} ({}%)".format(y + 1, round(100 * pca.explained_variance_ratio_[y],␣
↪1))
    )

    # Title
    plt.title("Cercle des corrélations (F{} et F{})".format(x + 1, y + 1))

    # Draw the circle
    an = np.linspace(0, 2 * np.pi, 100)
    plt.plot(np.cos(an), np.sin(an))  # Add a unit circle for scale

    # Axes and final display
    plt.axis("equal")
    plt.show(block=False)
```

Plot a first correlation graph (PC1 v PC2) :

[ ]:
```
correlation_graph(
    X_scaled,
    pca,
    dim=[0, 1],
)
```

Plot a 2nd correlation graph (PC2 v PC3)

[ ]:
```
correlation_graph(
    X_scaled,
    pca,
    dim=[0, 2],
)
```

Plot a 2nd correlation graph (PC1 v PC3)

[ ]:
```
correlation_graph(
    X_scaled,
```

```
    pca,
    dim=[1, 2],
)
```

1.6.5 Factorial planes

[ ]:
```
def factorial_planes(
    X_,
    pca,
    dim: list = [0, 1],
    labels: list = None,
    clusters: list = None,
    figsize: list = [12, 10],
    fontsize=14,
):
    """Display the Factorial plane projection of our data

    Positional arguments :
        - X_ : DataFrame | np.array : the scaled dataset
        - pca : PCA : a PCA instance already fitted

    Optional arguments :
        - dim : list or tuple : values x,y for the dimensions (PC) to plot.
            default :  [0,1] stands for F1, F2
        - labels : list or tuple or None: labels of our data such as countrie's␣
 ↪name for instance
        - clusters : list or tuple or None : clusters of our data such as 'A',␣
 ↪'B', 'C' for instance
        - figsize : list or tuple : The size in inches of our visualisation
            default : [12, 10]
        - fontsize : int  : The fontsize of our labels.
            default : 14
    """

    # Extract dimensions
    x, y = dim

    # Manage dtypes errors
    dtypes = (pd.DataFrame, np.ndarray, pd.Series, list, tuple, set)
    if not isinstance(labels, dtypes):
        labels = []
    if not isinstance(clusters, dtypes):
        clusters = []

    # Fig size
    fig, ax = plt.subplots(1, 1, figsize=figsize)
```

```
    # Display cluster with specific color IF NEEDED
    if len(clusters):
        sns.scatterplot(data=None, x=X_[:, x], y=X_[:, y], hue=clusters)
    else:
        sns.scatterplot(data=None, x=X_[:, x], y=X_[:, y])

    # Display explained variance ratio
    v1 = str(round(100 * pca.explained_variance_ratio_[x])) + " %"
    v2 = str(round(100 * pca.explained_variance_ratio_[y])) + " %"

    # Axes name, with % of explained inertia
    ax.set_xlabel(f"F{x+1} {v1}")
    ax.set_ylabel(f"F{y+1} {v2}")

    # Values for x max et y max
    x_max = np.abs(X_[:, x]).max() * 1.1
    y_max = np.abs(X_[:, y]).max() * 1.1

    # Set limits for x and y
    ax.set_xlim(left=-x_max, right=x_max)
    ax.set_ylim(bottom=-y_max, top=y_max)

    # Display horizontal and vertical lines
    plt.plot([-x_max, x_max], [0, 0], color="grey", alpha=0.8)
    plt.plot([0, 0], [-y_max, y_max], color="grey", alpha=0.8)

    # Labels and dots
    if len(labels):
        for i, (_x, _y) in enumerate(X_[:, [x, y]]):
            plt.text(
                _x, _y + 0.05, labels[i], fontsize=fontsize, ha="center",␣
↪va="center"
            )

    # Title and final display
    plt.title(f"Projection des individus (sur F{x+1} et F{y+1})")
    plt.show()
```

Plot a basic factorial plane :

[ ]:
```
factorial_planes(X_proj.values, pca, [0, 1])
```

Plot a factorial plane with size and labels :

[ ]:
```
factorial_planes(
    X_proj.values, pca, [0, 1], labels=df.zone.values, figsize=(20, 16),␣
↪fontsize=6
)
```