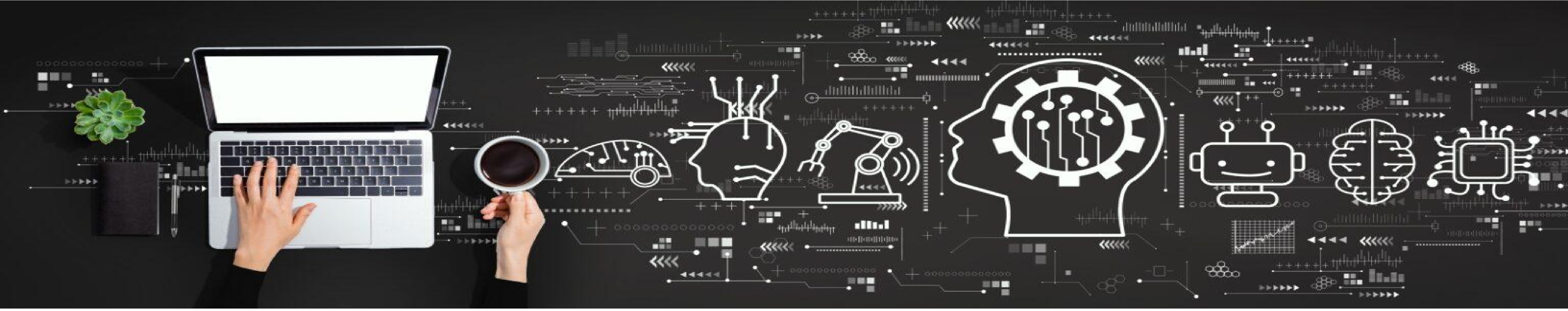


Data Preprocessing Feature Selection

Scikit Learn

Hichem Felouat
hichemfel@gmail.com

https://www.linkedin.com/in/hichemfelouat

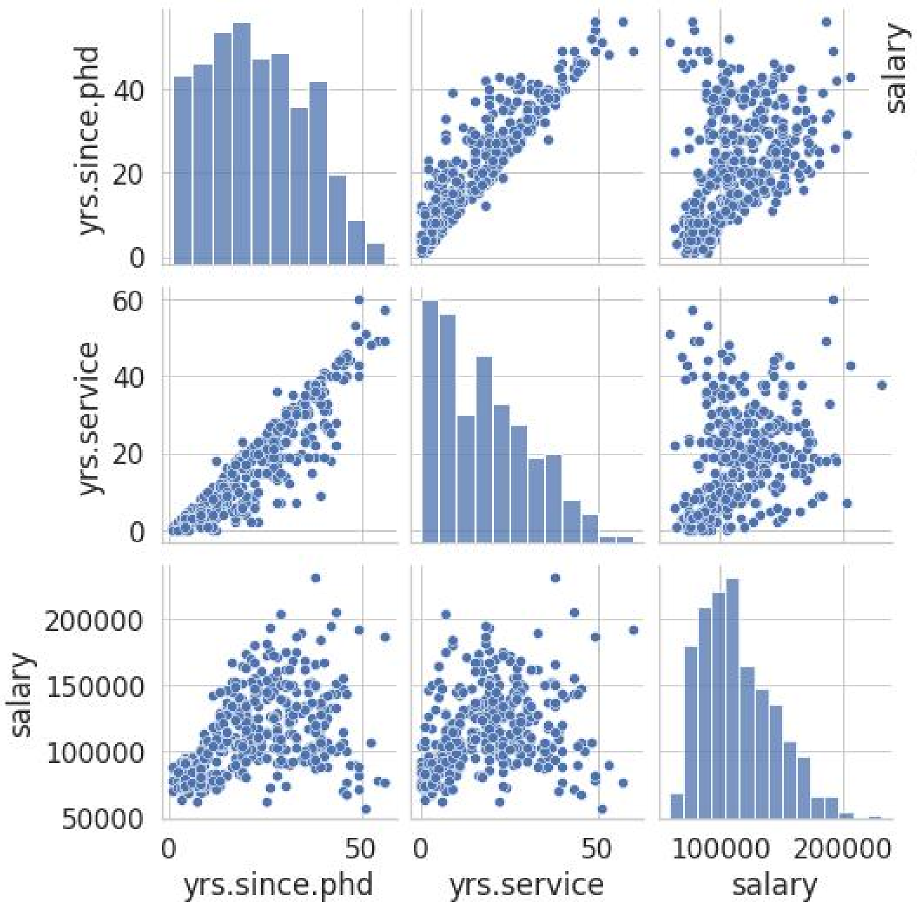
2. Data Preprocessing: Exploratory Data Analysis

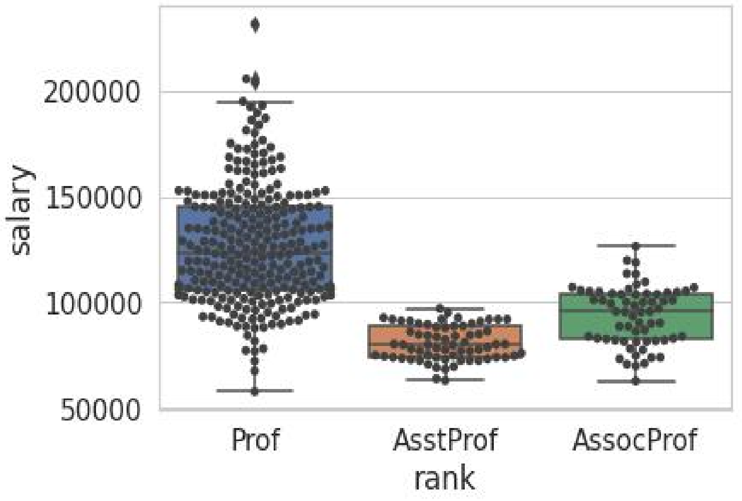
- **Exploratory Data Analysis or (EDA)** is **understanding the data sets** by summarizing their main characteristics often plotting them visually.

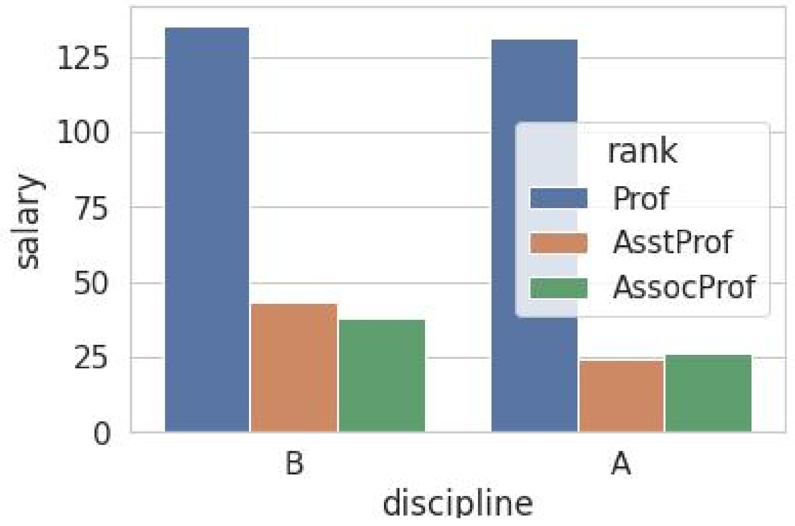
- This step is very important especially when we arrive at modeling the data **in order to apply Machine learning**.

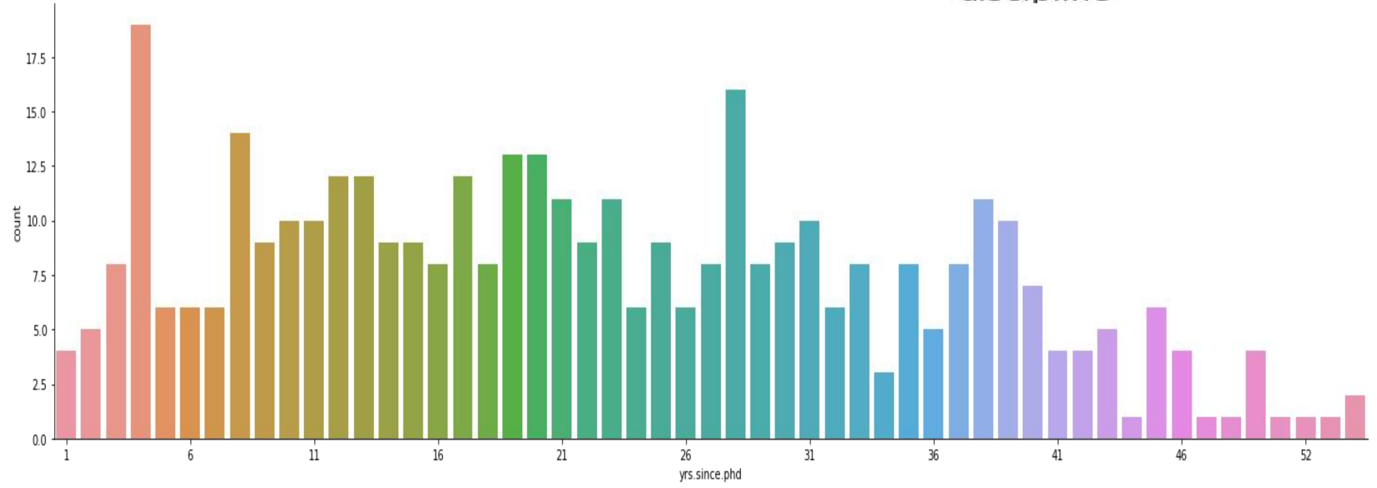
- Plotting in EDA consists of **Histograms**, **Box plot**, **Scatter plot** and many more. It often **takes much time** to explore the data. Through the process of EDA, we can ask to define **the problem statement** or definition on our data set which is very important.

2. Data Preprocessing: Exploratory Data Analysis

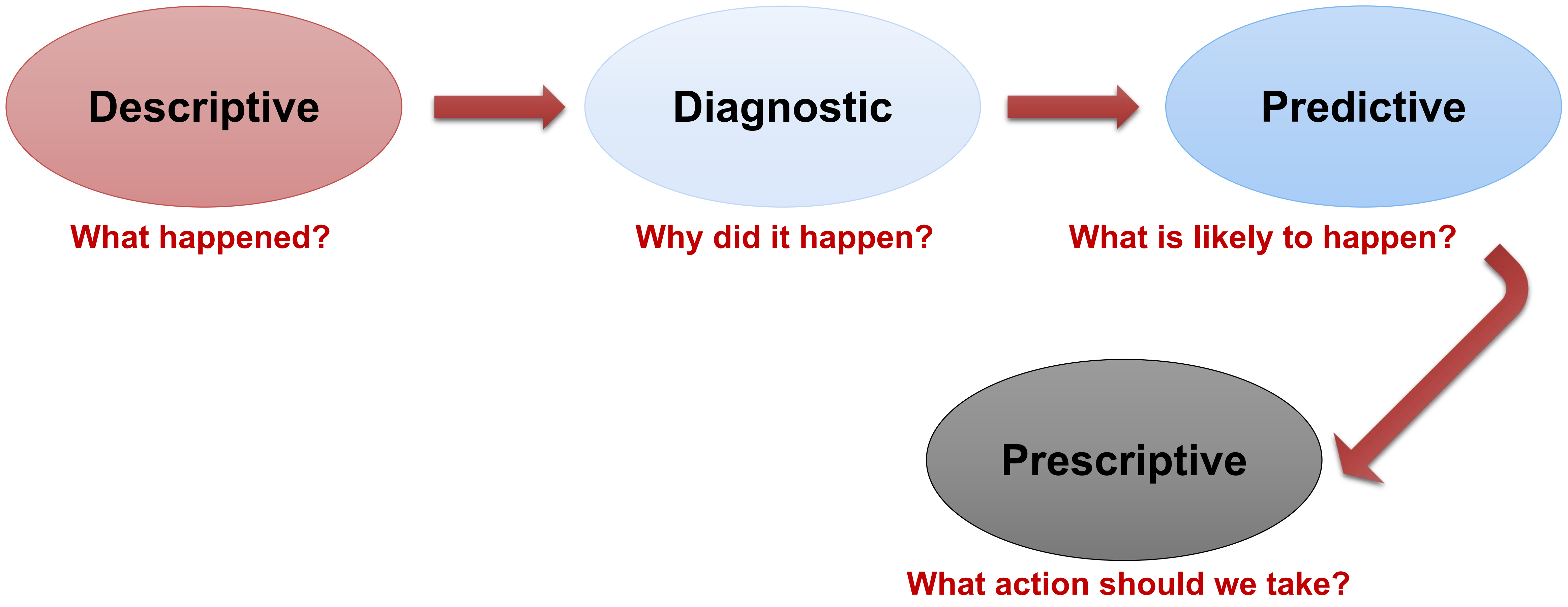




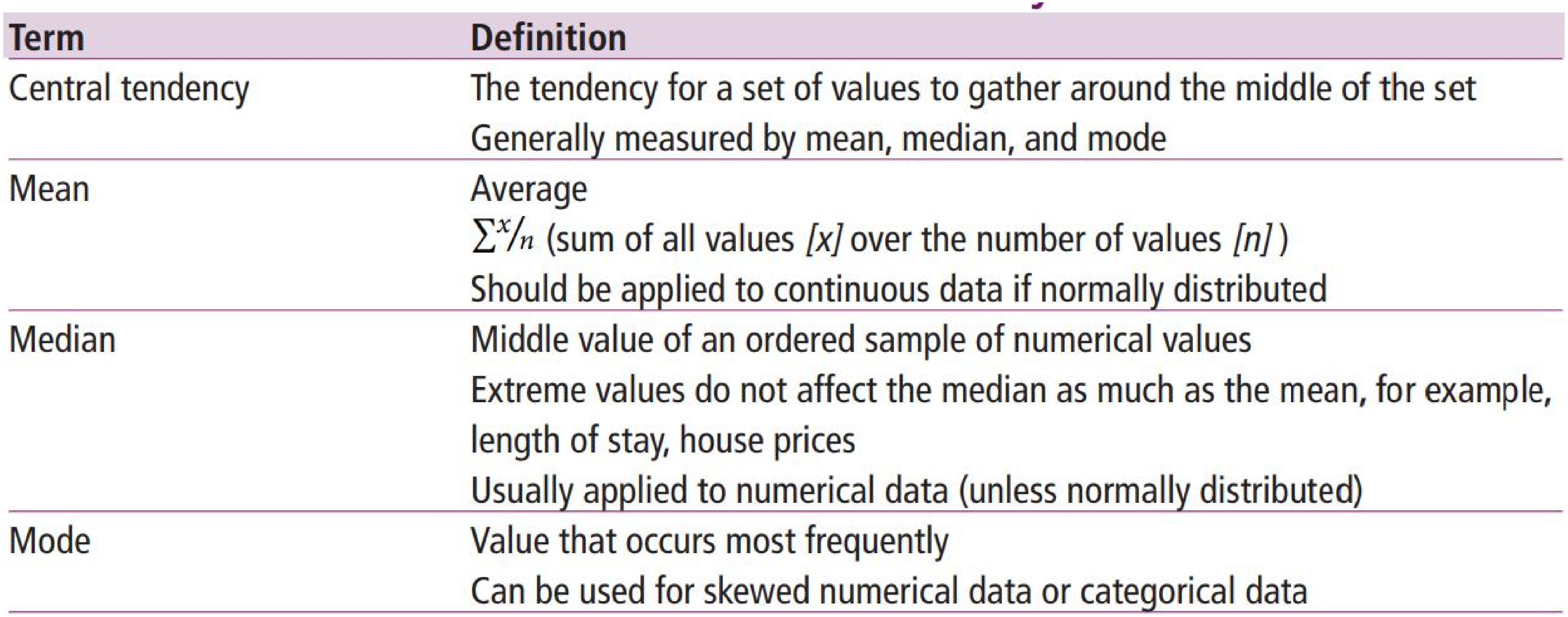




2. Data Preprocessing: Exploratory Data Analysis

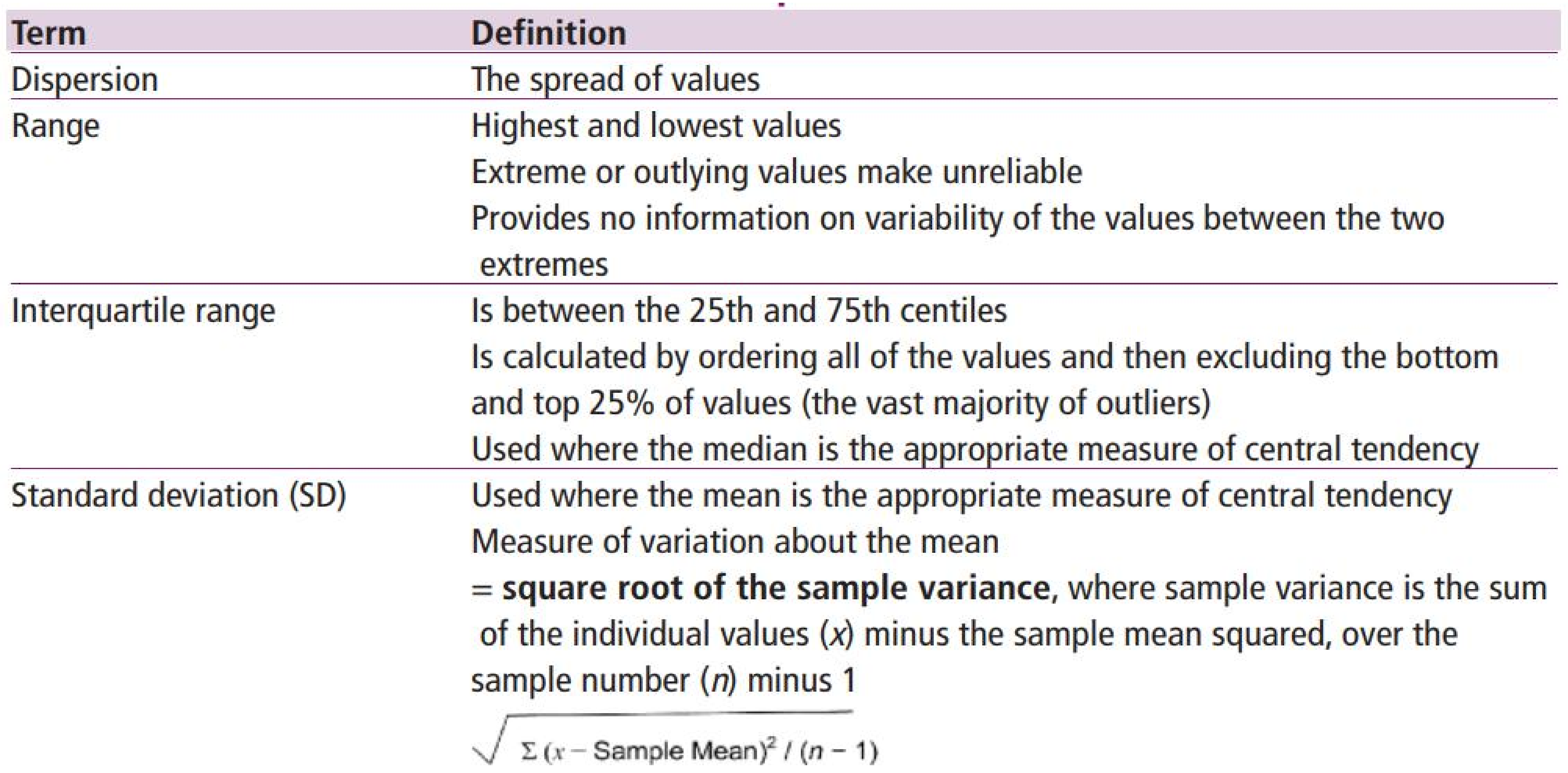


2. Data Preprocessing: Measures of Central Tendency



Term	Definition
Central tendency	The tendency for a set of values to gather around the middle of the set Generally measured by mean, median, and mode
Mean	Average $\sum x/n$ (sum of all values *[x]* over the number of values *[n]*) Should be applied to continuous data if normally distributed
Median	Middle value of an ordered sample of numerical values Extreme values do not affect the median as much as the mean, for example, length of stay, house prices Usually applied to numerical data (unless normally distributed)
Mode	Value that occurs most frequently Can be used for skewed numerical data or categorical data

2. Data Preprocessing: Measures of Dispersion



Term	Definition
Dispersion	The spread of values
Range	Highest and lowest values Extreme or outlying values make unreliable Provides no information on variability of the values between the two extremes
Interquartile range	Is between the 25th and 75th centiles Is calculated by ordering all of the values and then excluding the bottom and top 25% of values (the vast majority of outliers) Used where the median is the appropriate measure of central tendency
Standard deviation (SD)	Used where the mean is the appropriate measure of central tendency Measure of variation about the mean = **square root of the sample variance**, where sample variance is the sum of the individual values (*x*) minus the sample mean squared, over the sample number (*n*) minus 1 $\sqrt{\Sigma\,(x - \text{Sample Mean})^2 / (n - 1)}$

3. Feature Selection: Preprocessing

The **sklearn.preprocessing** package provides several common utility functions and transformer classes to change raw feature vectors into a representation that is more suitable for the downstream estimators.

1) ***Standardization***
2) ***Normalization***
3) ***Encoding categorical features***
4) ***Discretization***
5) ***Generating polynomial features***
6) ***Custom transformers***

3. Feature Selection: Preprocessing

1. Standardization: is a scaling technique where the **values are centered around the mean** with a unit **standard deviation**. This means that the mean of the attribute becomes zero and the resultant distribution has a unit standard deviation.

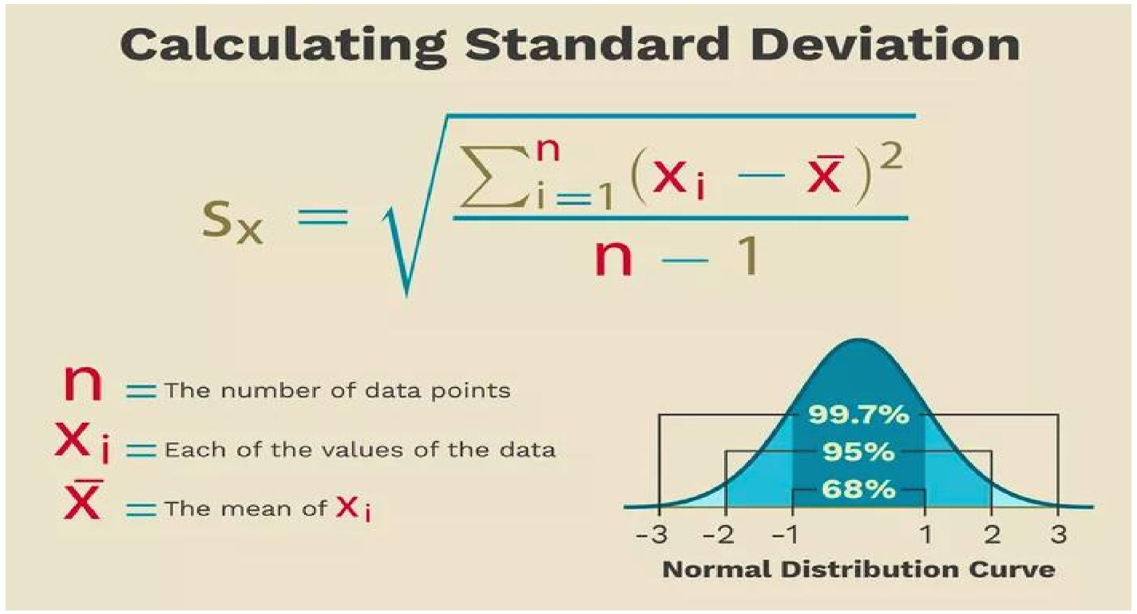
The standard score of a sample **x** is calculated as:

z = (x – u) / s

x = variable

u = mean

s = standard deviation



Calculating Standard Deviation
$$s_x = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$
n = The number of data points
x_i = Each of the values of the data
$\bar{x}$ = The mean of x_i
99.7%
95%
68%
-3
-2
-1
1
2
3
Normal Distribution Curve

3. Feature Selection: 1. Standardization - StandardScaler

```
from sklearn.preprocessing import StandardScaler
import numpy as np

X_train = np.array([[ 1., -1.,  2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])

scaler = StandardScaler().fit_transform(X_train)
print(scaler)
```

```
Out:
[[ 0.         -1.22474487  1.33630621]
 [ 1.22474487  0.         -0.26726124]
 [-1.22474487  1.22474487 -1.06904497]]
```

3. Feature Selection: 1. Standardization - Scaling Features to a Range

```
import numpy as np
from sklearn import preprocessing
X_train = np.array([[ 1., -1.,  2.], [ 2.,  0.,  0.], [ 0.,  1., -1.]])

# Here is an example to scale a data matrix to the [0, 1] range:
print("[0, 1] : \n")
min_0_max_1_scaler = preprocessing.MinMaxScaler()
X_train_min_0_max_1 = min_0_max_1_scaler.fit_transform(X_train)
print(X_train_min_0_max_1)

# between a given minimum and maximum value
print("min - max : \n")
min_max_scaler = preprocessing.MinMaxScaler(feature_range=(0, 10))
X_train_minmax = min_max_scaler.fit_transform(X_train)
print(X_train_minmax)

# scaling in a way that the training data lies within the range [-1, 1]
print("[-1, 1] : \n")
max_abs_scaler = preprocessing.MaxAbsScaler()
X_train_maxabs = max_abs_scaler.fit_transform(X_train)
print(X_train_maxabs)
```

3. Feature Selection: 1. Standardization - Scaling Data with Outliers

If your **data contains many outliers**, scaling using the mean and variance of the data is likely to not work very well. In these cases, you can use **robust_scale** **and** **RobustScaler** as drop-in replacements instead. They use more robust estimates for the center and range of your data.

```
import numpy as np
from sklearn import preprocessing

X_train = np.array([[ 1., -1.,  2.], [ 2.,  0.,  0.], [ 0.,  1., -1.]])
scaler = preprocessing.RobustScaler()
X_train_rob_scal = scaler.fit_transform(X_train)
print(X_train_rob_scal)
```

3. Feature Selection: 1. Standardization

```
# Save the scaler
from pickle import dump
dump(scaler, open("/content/scaler.pkl", "wb"))

# Load the scaler
from pickle import load
my_scaler = load(open("/content/scaler.pkl", "rb"))
scaler_result = my_scaler.transform(X_train)
print(scaler_result)
```

3. Feature Selection: 2. Normalization

Normalization is a scaling technique in which values are shifted and scaled so that **they end up ranging between 0 and 1**. It is also known as **Min-Max scaling**. It often used in **text classification** and **clustering** contexts.

```
from sklearn import preprocessing
import numpy as np

X = [[ 1., -1.,  2.], [ 2.,  0.,  0.], [ 0.,  1., -1.]]
X_normalized = preprocessing.normalize(X)
print(X_normalized)
```

3. Feature Selection: 2. Normalization

Normalization is good to use **when you know that the distribution of your data does not follow a Gaussian distribution**. This can be useful in algorithms that do not assume any distribution of the data like K-Nearest Neighbors and Neural Networks.

Standardization, on the other hand, can be helpful in cases where **the data follows a Gaussian distribution**.

3. Feature Selection: 3. Encoding Categorical Features

To convert categorical features to such integer codes, we can use the OrdinalEncoder. This estimator transforms each categorical feature to one new feature of integers (0 to n_categories - 1).

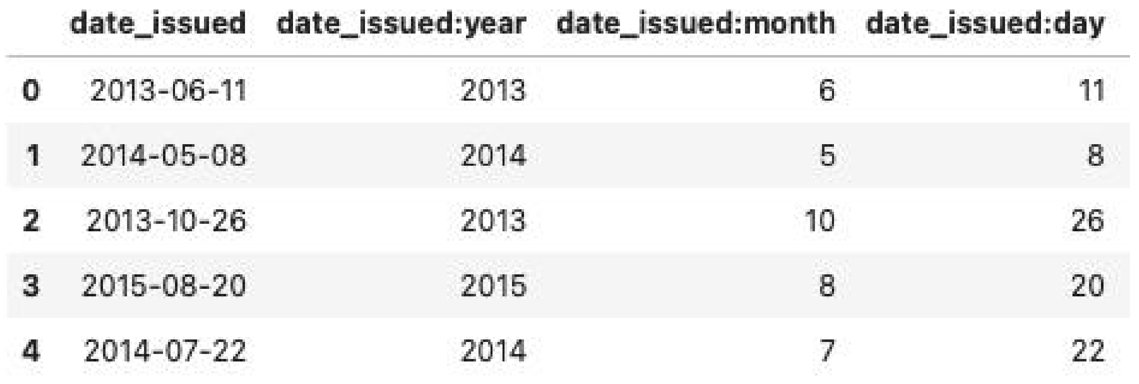
```
from sklearn import preprocessing
#genders = ['female', 'male']
#locations = ['from Africa', 'from Asia', 'from Europe', 'from US']
#browsers = ['uses Chrome', 'uses Firefox', 'uses Safari']

X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Safari'],
    ['female', 'from Asia', 'uses Firefox'], ['male', 'from Africa', 'uses Chrome']]
enc = preprocessing.OrdinalEncoder()
X_enc = enc.fit_transform(X)
print(X_enc)
```

```
[[1. 3. 2.]
 [0. 2. 2.]
 [0. 1. 1.]
 [1. 0. 0.]]
```

3. Feature Selection: 3. Encoding Categorical Features

Datetime Feature Engineering: we can extract the component of the date-time part **(year, quarter, month, day, day_of_week, day_of_year, week_of_year, time, hour, minute, second, day_part)** from the given date-time variable.

```
df["Date"] = pd.to_datetime(df["Date"])

df["year"]    = df["Date"].dt.year
df["month"] = df["Date"].dt.month
df["day"]      = df["Date"].dt.day
df["week_of_year"] = df["Date"].dt.weekofyear
df["day_of_year"]    = df["Date"].dt.dayofyear

df= df.drop(["Date"], axis=1)
print(df.info())
```



	date_issued	date_issued:year	date_issued:month	date_issued:day
0	2013-06-11	2013	6	11
1	2014-05-08	2014	5	8
2	2013-10-26	2013	10	26
3	2015-08-20	2015	8	20
4	2014-07-22	2014	7	22

3. Feature Selection: 4. Encoding Categorical Labels

Label Encoding (LabelEncoder) is a popular encoding technique for handling categorical variables. In this technique, each label is assigned a unique integer based on alphabetical orderingwith value between 0 and n_classes-1.

```
# Encoding Categorical Labels
from sklearn import preprocessing

labels = ["India", "US", "Japan", "US", "Japan"]
le = preprocessing.LabelEncoder()
new_labels = le.fit_transform(labels)

print(new_labels)

print("inverse_transform : \n",
                              le.inverse_transform([2, 0, 1]))
```

The country names do not have an **order** or **rank**. But, when label encoding is performed, the country names are ranked based on the alphabets. Due to this, there is a very high probability that the model captures the relationship between countries such as India < Japan < US.

3. Feature Selection: 4. Encoding Categorical Labels

One-Hot Encoding is another popular technique for treating categorical variables. It simply creates additional features based on the number of unique values in the categorical feature. Every unique value in the category will be added as a feature.

Country	Age	Salary
India	44	72000
US	34	65000
Japan	46	98000
US	35	45000
Japan	23	34000

Country	Age	Salary
0	44	72000
2	34	65000
1	46	98000
2	35	45000
1	23	34000

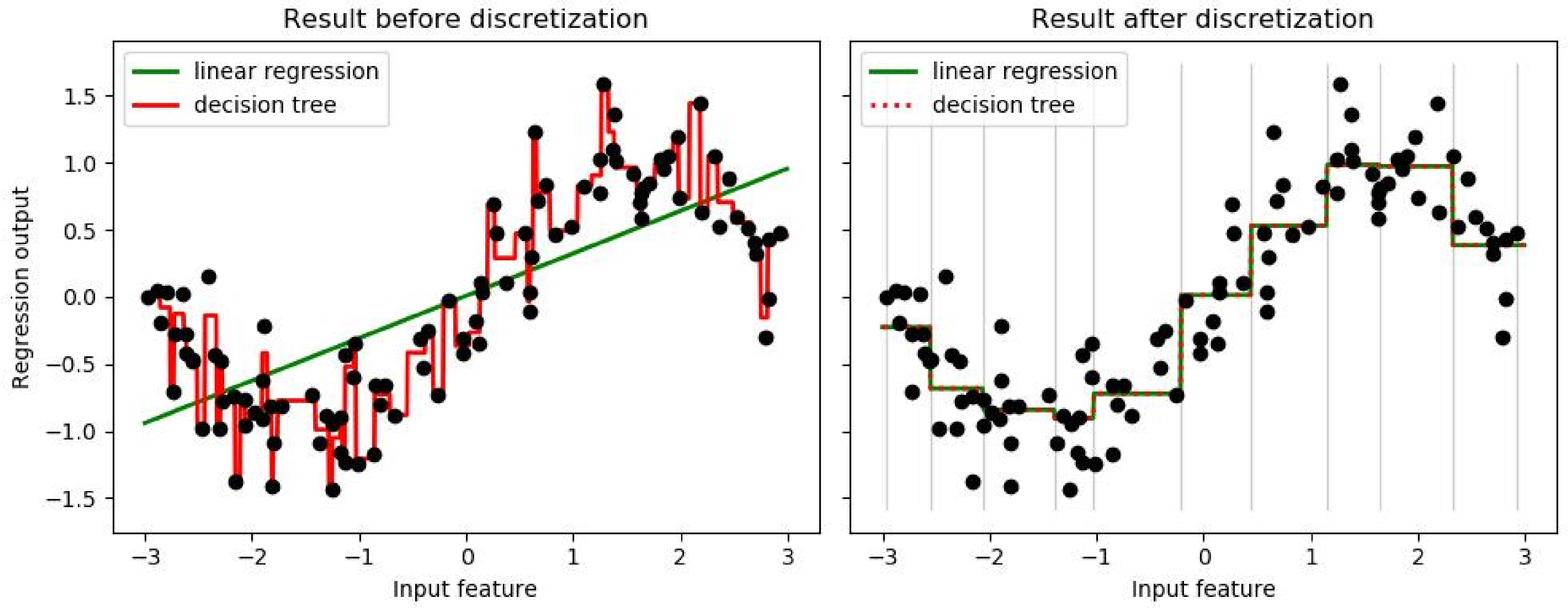
0	1	2	Age	Salary
1	0	0	44	72000
0	0	1	34	65000
0	1	0	46	98000
0	0	1	35	45000
0	1	0	23	34000

3. Feature Selection: 4. Encoding Categorical Labels

```
# Importing one hot encoder
from sklearn.preprocessing import OneHotEncoder
# Creating one hot encoder object
onehotencoder = OneHotEncoder()

# Reshape the 1-D country array to 2-D as fit_transform expects 2-D
and finally fit the object
X = onehotencoder.fit_transform(my_data.Country.values.reshape(-
                                                            1,1)).toarray()
print(X)
```

3. Feature Selection: 5. Discretization

Discretization (otherwise known as quantization or binning) provides a way to **partition continuous features into discrete values**.



3. Feature Selection: 5. Discretization

```
from sklearn import preprocessing
import numpy as np
X = np.array([[ -3., 5., 15 ],
              [  0., 6., 14 ],
              [  6., 3., 11 ]])
# 'onehot', 'onehot-dense', 'ordinal'
kbd = preprocessing.KBinsDiscretizer(n_bins=[3, 2, 2],
encode='ordinal')

X_kbd = kbd.fit_transform(X)
print(X_kbd)
```

3. Feature Selection: 5.1 Feature Binarization

```
from sklearn import preprocessing
import numpy as np
X = [[ 1., -1.,  2.],[ 2.,  0.,  0.],[ 0.,  1., -1.]]
binarizer = preprocessing.Binarizer()
X_bin = binarizer.fit_transform(X)
print(X_bin)

# It is possible to adjust the threshold of the binarizer:
binarizer_1 = preprocessing.Binarizer(threshold=1.1)
X_bin_1 = binarizer_1.fit_transform(X)
print(X_bin_1)
```

3. Feature Selection: 6. Generating Polynomial Features

Often it is useful to add complexity to the model by considering **nonlinear features** of the input data. A simple and common method to use is polynomial features, which can get features' high-order and interaction terms. It is implemented in PolynomialFeatures.

for 2 features :

$$(X_1, X_2) \text{ to } (1, X_1, X_2, X_1^2, X_1X_2, X_2^2)$$

3. Feature Selection: 6. Generating Polynomial Features

```
from sklearn import preprocessing
import numpy as np
X = np.arange(9).reshape(3, 3)
print(X)
poly = preprocessing.PolynomialFeatures(degree=3,
interaction_only=True)

X_poly = poly.fit_transform(X)
print(X_poly)
```

3. Feature Selection: 7. Custom Transformers

Often, you will want to convert an existing Python function into a transformer to assist in data cleaning or processing. You can implement a transformer from an arbitrary function with FunctionTransformer. **For example, to build a transformer that applies a log transformation in a pipeline, do:**

```
from sklearn import preprocessing
import numpy as np

transformer = preprocessing.FunctionTransformer(np.log1p,
validate=True)
X = np.array([[0, 1], [2, 3]])
X_tr = transformer.fit_transform(X)
print(X_tr)
```

3. Feature Selection: Text Feature

scikit-learn provides utilities for the most common ways to **extract numerical features** from text content, namely:

- **Tokenizing** strings and giving an integer **id** for each possible token, for instance by using white-spaces and punctuation as token separators.

- **Counting** the occurrences of tokens in each document.

- **Normalizing** and weighting with diminishing importance tokens that occur in the majority of samples / documents.

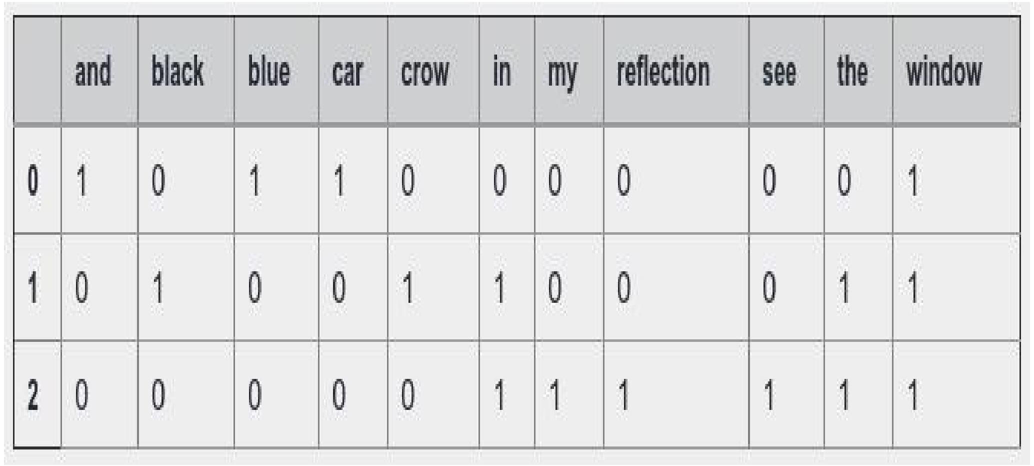
3. Feature Selection: Text Feature

A simple way we can convert text to numeric feature is via binary encoding. In this scheme, we create a vocabulary by looking at each distinct word in the whole dataset (corpus). For each document, the output of this scheme will be a vector of size N where N is the total number of words in our vocabulary. Initially all entries in the vector will be 0. If the word in the given document exists in the vocabulary then vector element at that position is set to 1.

CountVectorizer implements both tokenization and occurrence counting in a single class.

3. Feature Selection: Text Feature

```
from sklearn.feature_extraction.text import CountVectorizer
texts = [
    "blue car and blue window",
    "black crow in the window",
    "i see my reflection in the window"
]
vec = CountVectorizer(binary=True)
vec.fit(texts)
print([w for w in sorted(vec.vocabulary_.keys())])
X = vec.transform(texts).toarray()
print(X)

import pandas as pd
pd.DataFrame(vec.transform(texts).toarray(), columns=sorted(vec.vocabulary_.keys()))
```

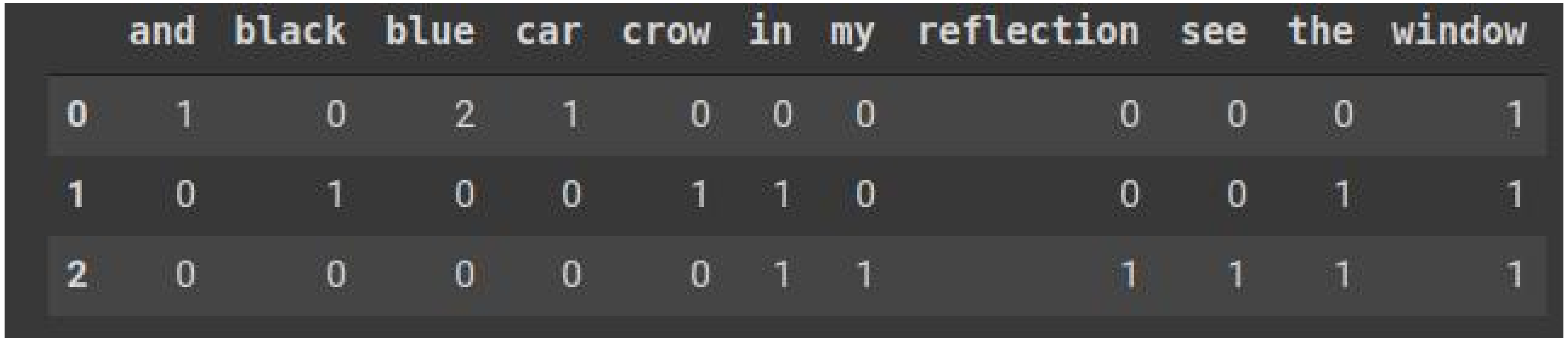


	and	black	blue	car	crow	in	my	reflection	see	the	window
0	1	0	1	1	0	0	0	0	0	0	1
1	0	1	0	0	1	1	0	0	0	1	1
2	0	0	0	0	0	1	1	1	1	1	1

3. Feature Selection: Text Feature

Counting is another approach to represent text as a numeric feature. It is similar to Binary scheme that we saw earlier but instead of just checking if a word exists or not, it also checks how many times a word appeared.

```
vec = CountVectorizer(binary=False)
```



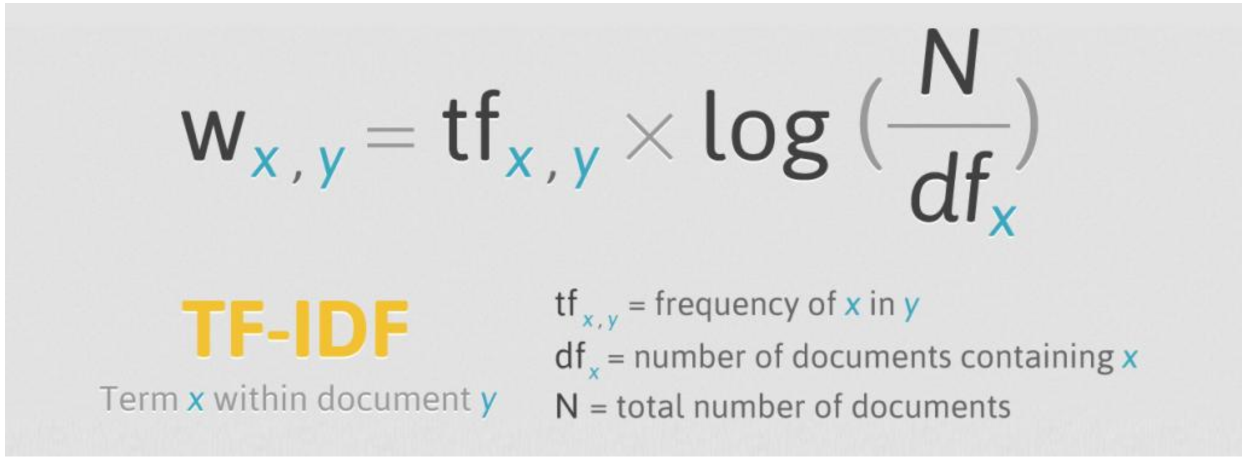
	and	black	blue	car	crow	in	my	reflection	see	the	window
0	1	0	2	1	0	0	0	0	0	0	1
1	0	1	0	0	1	1	0	0	0	1	1
2	0	0	0	0	0	1	1	1	1	1	1

3. Feature Selection: Text Feature

TF-IDF stands for **term frequency-inverse document frequency**. We saw that Counting approach assigns weights to the words based on their frequency and it’s obvious that frequently occurring words will have higher weights. But these words might not be important as other words. For example, let’s consider an article about Travel and another about Politics. Both of these articles will contain words like a, the frequently. But words such as flight, holiday will occur mostly in Travel and parliament, court etc. will appear mostly in Politics. Even though these words appear less frequently than the others, they are more important. TF-IDF assigns more weight to less frequently occurring words rather than frequently occurring ones. It is based on the assumption that less frequently occurring words are more important.

3. Feature Selection: Text Feature

```
from sklearn.feature_extraction.text import TfidfVectorizer
texts = [
    "blue car and blue window",
    "black crow in the window",
    "i see my reflection in the window"
]

vec = TfidfVectorizer()
vec.fit(texts)
print([w for w in sorted(vec.vocabulary_.keys())])
X = vec.transform(texts).toarray()
import pandas as pd
pd.DataFrame(vec.transform(texts).toarray(),
columns=sorted(vec.vocabulary_.keys()))
```

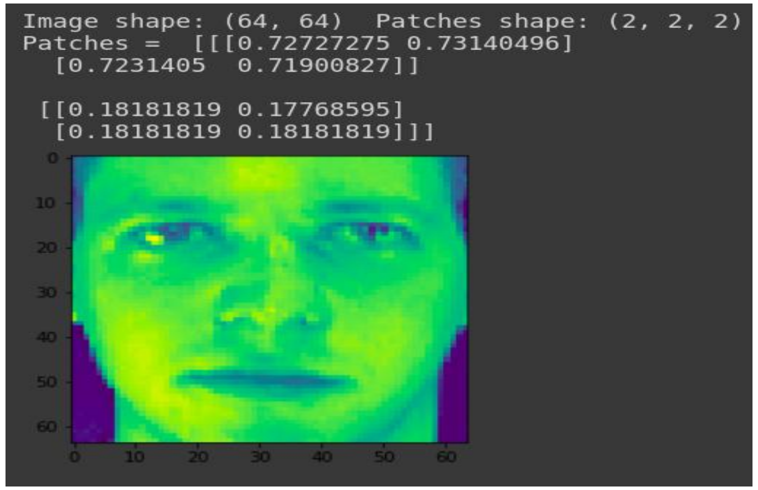

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

TF-IDF
Term x within document y

$tf_{x,y}$ = frequency of x in y
df_x = number of documents containing x
N = total number of documents

3. Feature Selection: Image Feature

```
# image.extract_patches_2d

from sklearn.feature_extraction import image
from sklearn.datasets import fetch_olivetti_faces
import matplotlib.pyplot as plt
import matplotlib.image as img

data = fetch_olivetti_faces()
plt.imshow(data.images[0])

# patches = image.extract_patches_2d(data.images[0], (3, 3),
max_patches=2,random_state=0)
patches = image.extract_patches_2d(data.images[0], (3, 3))
print("Image shape: ", data.images[0].shape, " Patches shape: ", patches.shape)
print("Patches : \n",len(patches.flatten()))
```



```
Image shape: (64, 64)  Patches shape: (2, 2, 2)
Patches =  [[[0.72727275 0.73140496]
  [0.7231405  0.71900827]]

 [[0.18181819 0.17768595]
  [0.18181819 0.18181819]]]
```

3. Feature Selection: Image Feature

```
import cv2
def hu_moments(image):
  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
  feature = cv2.HuMoments(cv2.moments(image)).flatten()
  return feature

def histogram(image,mask=None):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    hist = cv2.calcHist([image],[0],None,[256],[0,256])
    cv2.normalize(hist, hist)
    return hist.flatten()
```

3. Feature Selection: Image Feature

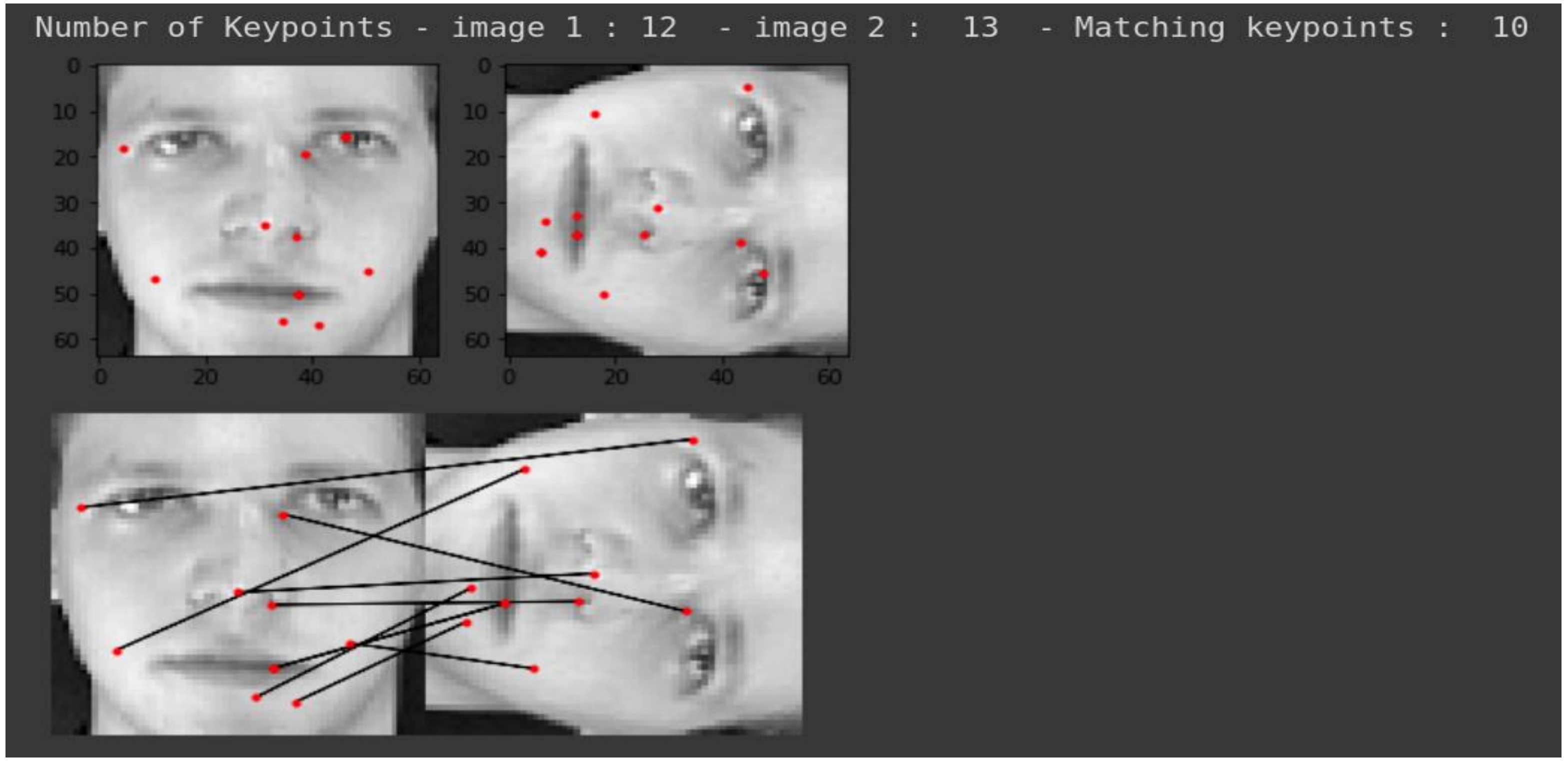
```
import mahotas
def haralick_moments(image):
  #image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
  image = image.astype(int)
  haralick = mahotas.features.haralick(image).mean(axis=0)
  return haralick

class ZernikeMoments:
        def __init__(self, radius):
                # store the size of the radius that will be
                # used when computing moments
                self.radius = radius

        def describe(self, image):
                # return the Zernike moments for the image
                return mahotas.features.zernike_moments(image, self.radius)
```

3. Feature Selection: Image Feature

```
import cv2
import mahotas
import numpy as np
from sklearn.datasets import fetch_olivetti_faces
import matplotlib.pyplot as plt

data = fetch_olivetti_faces()
plt.imshow(data.images[0])

hu_mot = hu_moments(data.images[0])
print("hu_mot : ", len(hu_mot),"\n",hu_mot)

hist = histogram(data.images[0])
print("hist : ", len(hist),"\n",hist)

haralick = haralick_moments(data.images[0])
print("haralick : ", len(haralick),"\n",haralick)

desc = ZernikeMoments(21)
zernike = desc.describe(data.images[0])
print("zernike : ", len(zernike),"\n",zernike)
```

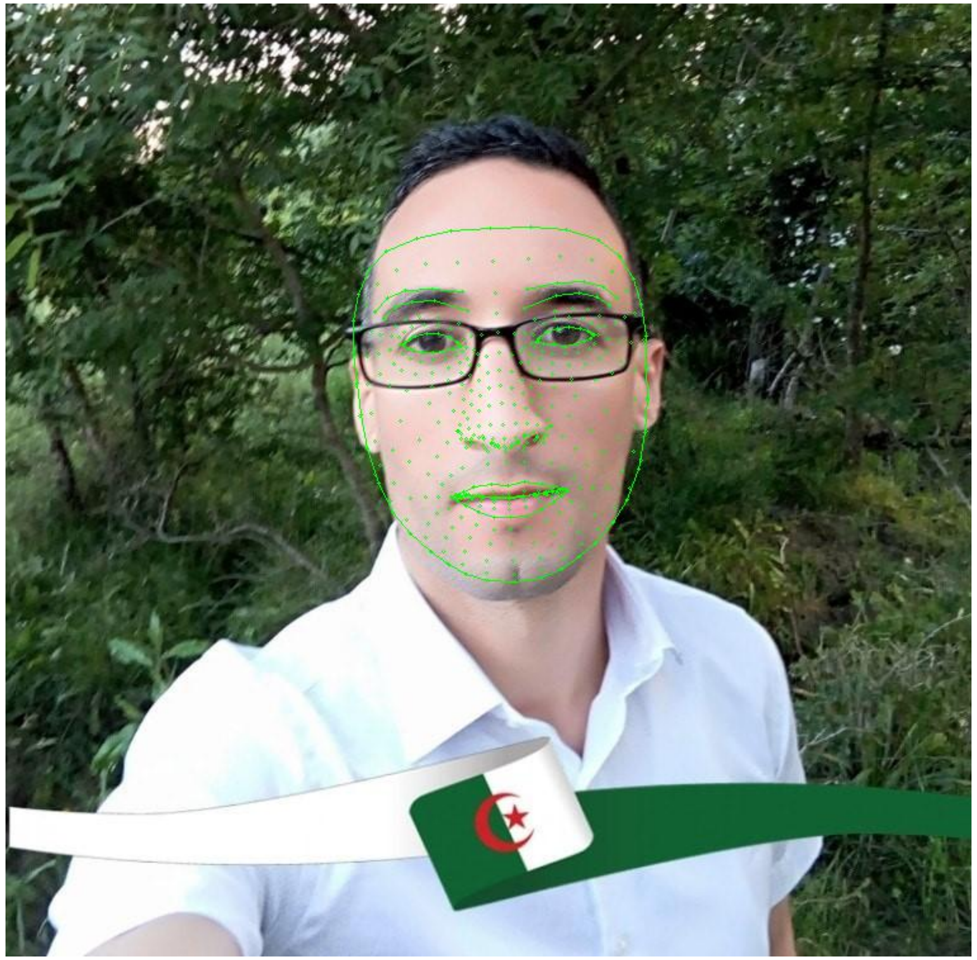
3. Feature Selection: Image Feature

```
from silx.opencl import sift
sift_ocl = sift.SiftPlan(template=img, devicetype="GPU")
keypoints = sift_ocl.keypoints(img)
```


Number of Keypoints - image 1 : 12 - image 2 : 13 - Matching keypoints : 10

3. Feature Selection: Image Feature

```
!pip install mediapipe
!pip install cvzone
# https://github.com/cvzone/cvzone


from cvzone.FaceMeshModule import FaceMeshDetector
import cv2

img = cv2.imread("/content/1.jpg")
detector = FaceMeshDetector(maxFaces=2)

img, faces = detector.findFaceMesh(img)
if faces:
    print(faces[0])

cv2_imshow(img)
```



Face Landmark

4. Dimensionality Reduction: Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a statistical method that creates new features or characteristics of data by analyzing the characteristics of the dataset. Essentially, the characteristics of the data are summarized or combined together. You can also conceive of Principal Component Analysis as "squishing" data down into just a few dimensions from much higher dimensions space.

4. Dimensionality Reduction: Principal Component Analysis (PCA)

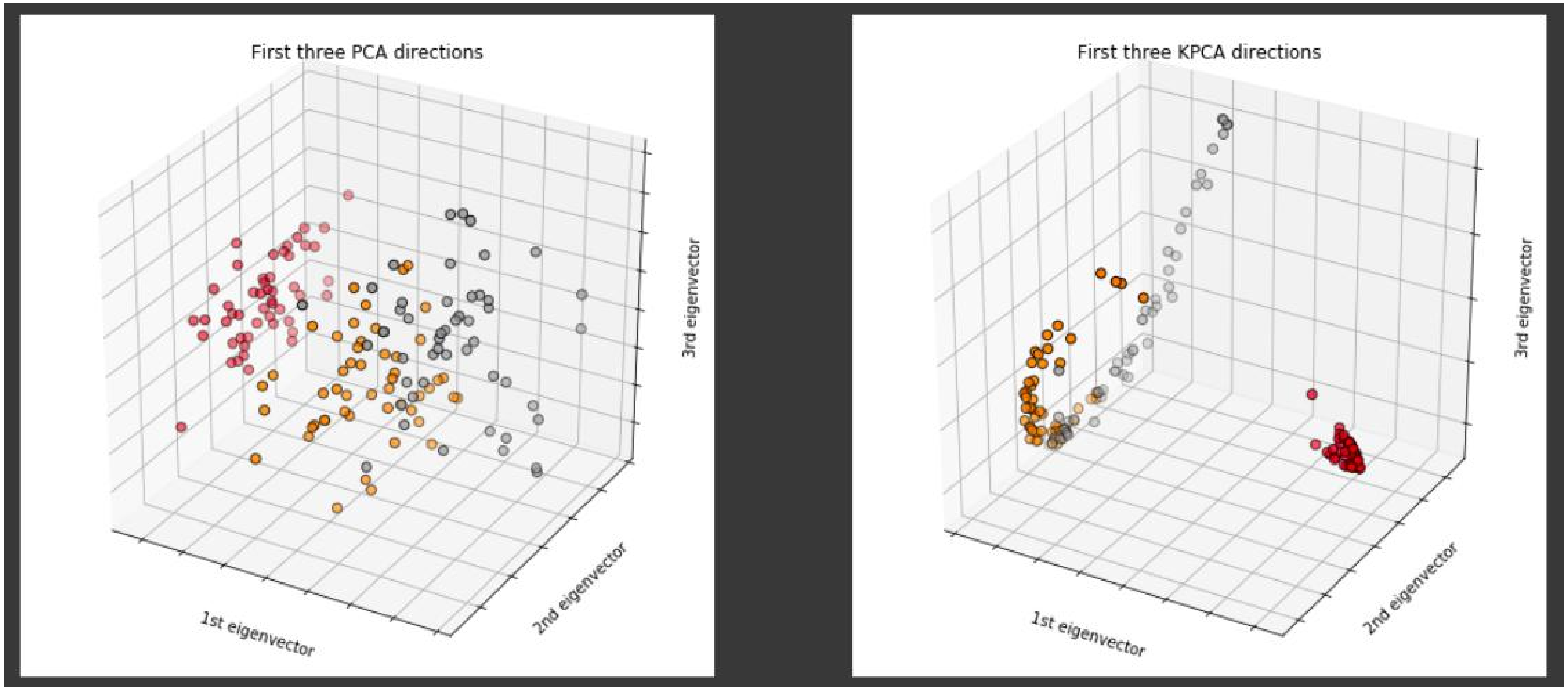
```
from sklearn import datasets
from sklearn.decomposition import PCA

dat = datasets.load_breast_cancer()
X, Y = dat.data, dat.target
print("Examples = ",X.shape ," Labels = ", Y.shape)

pca = PCA(n_components = 5)
X_pca = pca.fit_transform(X)
print("Examples_PCA = ",X_pca.shape ," Labels = ", Y.shape)
```

4. Dimensionality Reduction: Kernel Principal Component Analysis (KPCA)

Non-linear dimensionality reduction through the use of kernels.

```
from sklearn import datasets
from sklearn.decomposition import KernelPCA

dat = datasets.load_breast_cancer()
X, Y = dat.data, dat.target
print("Examples = ",X.shape ," Labels = ", Y.shape)
# kernel : "linear" | "poly" | "rbf" | "sigmoid" | "cosine" | "precomputed"
kpca = KernelPCA(n_components=7, kernel='rbf')
X_kpca = kpca.fit_transform(X)
print("Examples = ",X_kpca.shape ," Labels = ", Y.shape)
```

4. Dimensionality Reduction: PCA VS KPCA



4. Dimensionality Reduction: Linear Discriminant Analysis (LDA)

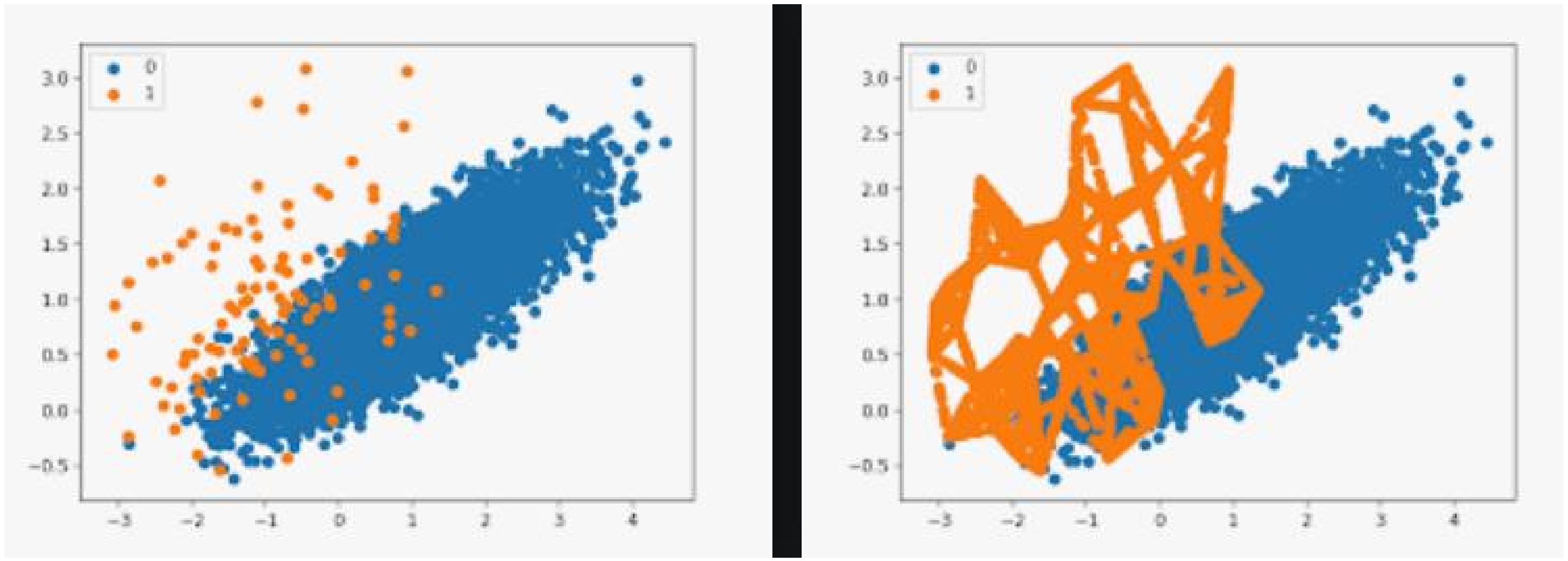
In case of **uniformly distributed data**, LDA almost always performs better than PCA. However if the data is highly skewed (irregularly distributed) then it is advised to use PCA since LDA can be biased towards the majority class.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit(X, Y).transform(X)
print("Examples = ",X_lda.shape ," Labels = ", Y.shape)
```

5. Having an Imbalanced Dataset?

- The learning phase and the subsequent prediction of machine learning algorithms can be affected by the problem of imbalanced data set. The balancing issue corresponds to the difference of the number of samples in the different classes.

- imbalanced-learn is a python package offering a number of re-sampling techniques commonly used in datasets showing strong between-class imbalance. It is compatible with scikit-learn and is part of scikit-learn-contrib projects.



5. Having an Imbalanced Dataset?

```
from imblearn.over_sampling import RandomOverSampler, SMOTE, ADASYN, BorderlineSMOTE,
SMOTENC

"""
ros = RandomOverSampler(random_state=0)
X_resampled, y_resampled = ros.fit_resample(X, y)
"""

X_resampled, y_resampled = SMOTE().fit_resample(X, y)

# X_resampled, y_resampled = ADASYN().fit_resample(X, y)

# X_resampled, y_resampled = BorderlineSMOTE().fit_resample(X, y)

"""
smote_nc = SMOTENC(categorical_features=[0], random_state=0) # categorical_features: Specified
which features are categorical
X_resampled, y_resampled = smote_nc.fit_resample(X, y)
"""
```

6. Training and Test Sets: Splitting Data

```
from sklearn.model_selection import train_test_split
from sklearn import datasets
dat = datasets.load_iris()
X = dat.data
Y = dat.target
print("Examples = ",X.shape ," Labels = ", Y.shape)
# stratify : If not None, data is split in a stratified fashion, using
this as the class labels.
X_train, X_test, Y_train, Y_test = train_test_split(X,
            Y, test_size= 0.20, random_state=100, stratify=Y)
print("X_train = ",X_train.shape ," Y_test = ", Y_test.shape)
```

6. Training and Test Sets: Splitting Data

```
import pandas as pd
from sklearn.model_selection import train_test_split

dataframe = pd.read_csv("Iris_Dataset.csv")
# split into input and output elements
dataframe["species"] = dataframe["species"].map({"Iris-setosa":0,"Iris-versicolor":1,
"Iris-virginica":2})
X = dataframe.drop(["species"],axis=1).values
Y = dataframe["species"].values
print("X: ",X.shape, " Y: ",y.shape)

# split into train test sets
X_train, X_test, Y_train, Y_test = train_test_split(X,
            Y, test_size= 0.20, random_state=100, stratify=Y)

print("X_train = ",X_train.shape ," Y_train = ", Y_train.shape)
print("X_test  = ",X_test.shape ," Y_test = ", Y_test.shape)
```

Thank you for your attention

Hichem Felouat …