Lab 8: Data Preprocessing

This tutorial uses standard python libraries used for data processing. The lab also introduces some basic accuracy measures used for performance evaluation purpose.

In [1]:
```
import pandas as pd, scipy, numpy as np
import sklearn.preprocessing
import sklearn.impute

ds = pd.read_excel("Job_Scheduling.xlsx")
```

Data Pre-processing

Data scientists come across many datasets and not all of them may be well formatted or noise free. While doing any kind of analysis with data it is important to clean it, as raw data can be highly unstructured with noise or missing data or data that is varying in scales which makes it hard to extract useful information. Pre-processing refers to the transformations applied to our data before feeding it to any machine learning algorithm. It is a technique that is used to convert the raw data into a clean data set. In other words, whenever the data is gathered from different sources it is collected in raw format which is not feasible for the analysis

In [2]:
```
print(ds)
```

```
    Job Id  Burst time  Arrival Time  Preemptive  Resources
0      334       179.0        0.6875         1.0        4.0
1      234       340.0        0.7800         0.0        4.0
2      138       143.0        0.9150         1.0        4.0
3      463       264.0           NaN         0.0        5.0
4      283       216.0        0.5550         0.0        6.0
5       88        36.0        0.6625         0.0        5.0
6      396       128.0        0.1975         1.0        NaN
7      470       203.0        0.9875         1.0        4.0
8      335       271.0        0.0275         0.0        3.0
9      272       399.0        0.2150         NaN        3.0
10     237         NaN        0.4825         1.0        4.0
11     318       311.0        0.5675         1.0        1.0
12      84       111.0        0.2725         1.0        2.0
13     311        87.0           NaN         0.0        7.0
14     163       103.0        0.4600         0.0        5.0
15     453       213.0        0.0775         1.0        1.0
16     176       251.0        0.7050         0.0        6.0
17     449        49.0        0.2550         1.0        4.0
18      11       168.0        0.3175         1.0        7.0
```

In [3]:
```
x = ds.iloc[:,0:4].values #Extracting all the entries from column 0-4 from the
np.set_printoptions(precision=3)
print(x)
```

```
[[3.340e+02 1.790e+02 6.875e-01 1.000e+00]
 [2.340e+02 3.400e+02 7.800e-01 0.000e+00]
 [1.380e+02 1.430e+02 9.150e-01 1.000e+00]
 [4.630e+02 2.640e+02       nan 0.000e+00]
 [2.830e+02 2.160e+02 5.550e-01 0.000e+00]
 [8.800e+01 3.600e+01 6.625e-01 0.000e+00]
 [3.960e+02 1.280e+02 1.975e-01 1.000e+00]
 [4.700e+02 2.030e+02 9.875e-01 1.000e+00]
 [3.350e+02 2.710e+02 2.750e-02 0.000e+00]
 [2.720e+02 3.990e+02 2.150e-01       nan]
 [2.370e+02       nan 4.825e-01 1.000e+00]
 [3.180e+02 3.110e+02 5.675e-01 1.000e+00]
 [8.400e+01 1.110e+02 2.725e-01 1.000e+00]
 [3.110e+02 8.700e+01       nan 0.000e+00]
 [1.630e+02 1.030e+02 4.600e-01 0.000e+00]
 [4.530e+02 2.130e+02 7.750e-02 1.000e+00]
 [1.760e+02 2.510e+02 7.050e-01 0.000e+00]
 [4.490e+02 4.900e+01 2.550e-01 1.000e+00]
 [1.100e+01 1.680e+02 3.175e-01 1.000e+00]]
```

Pandas iloc function

iloc returns a Pandas Series when one row is selected, and a Pandas DataFrame when multiple rows are selected, or if any column in full is selected.

In [4]:
```
y=ds.iloc[:,4].values     #Extracting all entries from column 4 from the 2D ar
print(y)
```

```
[ 4.  4.  4.  5.  6.  5. nan  4.  3.  3.  4.  1.  2.  7.  5.  1.  6.  4.
  7.]
```

Dealing with Missing Values

Machine learning models cannot accommodate missing fields in the data they are provided with. So, the missing fields must be filled with values that will not affect the variance of the data and make it less noisy. A basic strategy to use incomplete datasets is to discard entire rows and/or columns containing missing values. However, this comes at the price of losing data which may be valuable (even though incomplete). A better strategy is to impute the missing values, i.e., to infer them from the known part of the data.

Note: More details on the vast majority of Imputers available for different data types can be accessed on https://scikit-learn.org/stable/modules/impute.html#impute (https://scikit-learn.org/stable/modules/impute.html#impute)

The SimpleImputer class provides basic strategies for imputing missing values. Missing values can be imputed with a provided constant value, or using the statistics (mean, median or most frequent) of each column in which the missing values are located.

We have some missing fields in the data denoted by “nan” which is an acronym for “not a number”.

The following snippet demonstrates how to replace missing values, encoded as np.nan, using the mean value of the columns (axis 0) that contain the missing values:

Implementing the Transform

fit() :

The fit() method is a standard method used in many machine learning algorithms. Its primary purpose is to analyze the training data and learn the parameters needed for the model. These parameters could include coefficients, mean, standard deviation, or any other internal variables required for the model to make predictions on new data. For example, in the case of a scaler (like MinMaxScaler or StandardScaler), the fit() method computes the minimum and maximum values or mean and standard deviation of the training data. These computed parameters are then stored within the scaler object.

transform() :

The transform() method is applied to the input data using the parameters learned by the fit() method. It is used to apply the transformation to the training data, testing data, or any new data that the model will process. Continuing with the scaler example, the transform() method will use the parameters (e.g., min and max values for MinMaxScaler) obtained during the fit() phase to scale the features of the data within the specified range.

fit_transform() :

The fit_transform() method is a convenience method often provided in libraries like scikit-learn. It combines the fit() and transform() steps into a single operation. This can be more efficient than calling fit() and then transform() separately, especially for large datasets, as it avoids redundant computations. It fits the model and transforms the data in a single pass. Using the scaler example, fit_transform() computes the scaling parameters from the training data and immediately applies the transformation to that data in one go.

In [5]:
```
# Filling missing Values

from sklearn.impute import SimpleImputer
imp = SimpleImputer(missing_values=np.nan, strategy='mean')

X = imp.fit_transform(x)
print(X)
```

```
[[3.340e+02 1.790e+02 6.875e-01 1.000e+00]
 [2.340e+02 3.400e+02 7.800e-01 0.000e+00]
 [1.380e+02 1.430e+02 9.150e-01 1.000e+00]
 [4.630e+02 2.640e+02 4.803e-01 0.000e+00]
 [2.830e+02 2.160e+02 5.550e-01 0.000e+00]
 [8.800e+01 3.600e+01 6.625e-01 0.000e+00]
 [3.960e+02 1.280e+02 1.975e-01 1.000e+00]
 [4.700e+02 2.030e+02 9.875e-01 1.000e+00]
 [3.350e+02 2.710e+02 2.750e-02 0.000e+00]
 [2.720e+02 3.990e+02 2.150e-01 5.556e-01]
 [2.370e+02 1.929e+02 4.825e-01 1.000e+00]
 [3.180e+02 3.110e+02 5.675e-01 1.000e+00]
 [8.400e+01 1.110e+02 2.725e-01 1.000e+00]
 [3.110e+02 8.700e+01 4.803e-01 0.000e+00]
 [1.630e+02 1.030e+02 4.600e-01 0.000e+00]
 [4.530e+02 2.130e+02 7.750e-02 1.000e+00]
 [1.760e+02 2.510e+02 7.050e-01 0.000e+00]
 [4.490e+02 4.900e+01 2.550e-01 1.000e+00]
 [1.100e+01 1.680e+02 3.175e-01 1.000e+00]]
```

Reshaping your class attribute

A numpy imputer expects a 2D array as input. In order to impute your class attribute, you need to convert it into a 2D array first. This can be done by reshaping your data either using array.reshape(-1, 1) if your data has a single feature or array.reshape(1, -1) if it contains a single sample.

In [6]:
```
# As highlighted earlier y holds the shape of a 1D array, where as imputation
Y = y.reshape(-1,1) #reshaping the numpy array with -1 parameter (numpy is to
Y = imp.fit_transform(Y)
Y = Y.reshape(-1)   # putting y bank in its original shape
print(Y)
```

```
[4.    4.    4.    5.    6.    5.    4.167 4.    3.    3.    4.    1.
 2.    7.    5.    1.    6.    4.    7.   ]
```

The SimpleImputer class also supports categorical data represented as string values or pandas categoricals when using the 'most_frequent' or 'constant' strategy:

In [7]:
```
df = pd.DataFrame([["a", "x"],
                   [np.nan, "y"],
                   ["a", np.nan],
                   ["b", "y"]], dtype="category")

imp = SimpleImputer(strategy="most_frequent")
print(imp.fit_transform(df))
```

```
[['a' 'x']
 ['a' 'y']
 ['a' 'y']
 ['b' 'y']]
```

The following snippet demonstrates how to replace missing values, encoded as np.nan, using the mean feature value of the two nearest neighbors of samples with missing values:

In [8]:
```
from sklearn.impute import KNNImputer
nan = np.nan
P = [[1, 2, nan], [3, 4, 3], [nan, 6, 5], [8, 8, 7]]
imputer = KNNImputer(n_neighbors=2, weights="uniform")
imputer.fit_transform(P)
```

Out[8]:
```
array([[1. , 2. , 4. ],
       [3. , 4. , 3. ],
       [5.5, 6. , 5. ],
       [8. , 8. , 7. ]])
```

Rescaling the Variables

When the data is comprised of attributes with varying scales, many machine learning algorithms can benefit from rescaling the attributes to all have the same scale. It is also useful for algorithms that weight inputs like regression and neural networks and algorithms that use distance measures like K-Nearest Neighbors. Data can be rescaled using scikit-learn using the MinMaxScaler class.Scaling can be done between a given minimum and maximum value, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This can be achieved using MinMaxScaler or MaxAbsScaler, respectively.

The motivation to use this scaling include robustness to very small standard deviations of features and preserving zero entries in sparse data.

In [9]:
```
# Rescaling Burst Time
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0,1))
rescaledX = scaler.fit_transform(X[:,1].reshape(-1,1)) #picking column 1 for r
np.set_printoptions(precision=3)
X[:,1] = rescaledX.reshape(1,-1)
print(X[:,1])
```

```
[0.394 0.837 0.295 0.628 0.496 0.    0.253 0.46  0.647 1.    0.432 0.758
 0.207 0.14  0.185 0.488 0.592 0.036 0.364]
```

Feature scaling (also known as data normalization) is the method used to standardize the range of features of data. Since, the range of values of data may vary widely, it becomes a necessary step in data preprocessing while using machine learning algorithms. Scaling is important in the algorithms such as support vector machines (SVM) and k-nearest neighbors (KNN) where distance between the data points is important. For example, in the dataset containing prices of products; without scaling, KNN might treat 1 USD equivalent to 1 PKR though 1 USD = 20 PKR. It is important to note that scaling does not change the data distribution.

Normalizing the Values of the Varaibles

The point of normalization is to change your observations so that they can be described as a normal distribution. Normal distribution (Gaussian distribution), also known as the bell curve, is a specific statistical distribution where a roughly equal observations fall above and below the mean, the mean and the median are the same, and there are more observations closer to the mean.

The preprocessing module provides a utility class Normalizer that implements the same operation using the Transformer API:

In [10]:
```
#Normalizing data
from sklearn.preprocessing import Normalizer
scaler = Normalizer().fit(X)
normalizedX = scaler.transform(X)
normalizedX
```

Out[10]:
```
array([[1.000e+00, 1.179e-03, 2.058e-03, 2.994e-03],
       [1.000e+00, 3.579e-03, 3.333e-03, 0.000e+00],
       [9.999e-01, 2.136e-03, 6.630e-03, 7.246e-03],
       [1.000e+00, 1.357e-03, 1.037e-03, 0.000e+00],
       [1.000e+00, 1.752e-03, 1.961e-03, 0.000e+00],
       [1.000e+00, 0.000e+00, 7.528e-03, 0.000e+00],
       [1.000e+00, 6.400e-04, 4.987e-04, 2.525e-03],
       [1.000e+00, 9.788e-04, 2.101e-03, 2.128e-03],
       [1.000e+00, 1.932e-03, 8.209e-05, 0.000e+00],
       [1.000e+00, 3.676e-03, 7.904e-04, 2.042e-03],
       [1.000e+00, 1.824e-03, 2.036e-03, 4.219e-03],
       [1.000e+00, 2.382e-03, 1.785e-03, 3.145e-03],
       [9.999e-01, 2.459e-03, 3.244e-03, 1.190e-02],
       [1.000e+00, 4.518e-04, 1.544e-03, 0.000e+00],
       [1.000e+00, 1.132e-03, 2.822e-03, 0.000e+00],
       [1.000e+00, 1.076e-03, 1.711e-04, 2.207e-03],
       [1.000e+00, 3.365e-03, 4.006e-03, 0.000e+00],
       [1.000e+00, 7.976e-05, 5.679e-04, 2.227e-03],
       [9.949e-01, 3.289e-02, 2.872e-02, 9.045e-02]])
```

Data Encoding

In many Machine-learning or Data Science activities, the data set might contain text or categorical values (basically non-numerical values). For example, color feature having values like red, orange, blue, white etc. Meal plan having values like breakfast, lunch, snacks, dinner,

tea etc. Few algorithms such as decision-trees can handle categorical values very well but most of the algorithms expect numerical values to achieve state-of-the-art results

Label Encoding

This approach is very simple and it involves converting each value in a column to a number

In [11]:
```
ds = pd.read_excel('weatherTemp.xlsx')
# splitting the non-class and class attributes
x = ds.iloc[:, 0:2].values  #The iloc indexer for Pandas Dataframe is used for
y = ds.iloc[:, 2].values #integer-location based indexing / selection by posit
print(x)
print(y)
```

```
[['sunny' 'hot']
 ['sunny' 'hot']
 ['overcast' 'hot']
 ['rainy' 'mild']
 ['rainy' 'cool']
 ['rainy' 'cool']
 ['overcast' 'cool']
 ['sunny' 'mild']
 ['sunny' 'cool']
 ['rainy' 'mild']
 ['sunny' 'mild']
 ['overcast' 'mild']
 ['overcast' 'hot']
 ['rainy' 'mild']]
['no' 'no' 'yes' 'yes' 'yes' 'no' 'yes' 'no' 'yes' 'yes' 'yes' 'yes' 'yes'
 'no']
```

In [12]:
```
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()

#Converting String Labels into numbers one column at a time
x[:,0]=encoder.fit_transform(x[:,0])
x[:,1] = encoder.fit_transform(x[:,1])
y = encoder.fit_transform(y)

#Observing the transformed data set
print("Weather:", x[:,0])
print("Temp:", x[:,1])
print("Play:", y)
```

```
Weather: [2 2 0 1 1 1 0 2 2 1 2 0 0 1]
Temp: [1 1 1 2 0 0 0 2 0 2 2 2 1 2]
Play: [0 0 1 1 1 0 1 0 1 1 1 1 1 0]
```