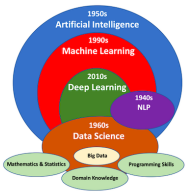
Department of Data Science

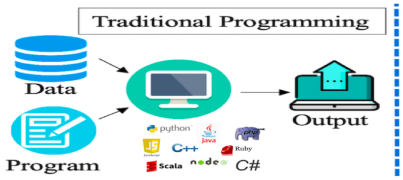
Course: Tools and Techniques for Data Science

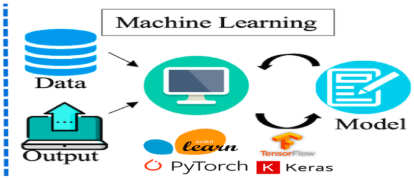
Instructor: Muhammad Arif Butt, Ph.D.

Lecture 6.10 (Data Preprocessing: Feature Scaling)

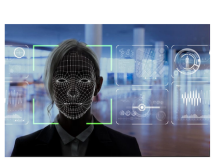
Open in Colab
(https://colab.research.google.com/github/arifpucit/data-science/blob/master/Section-4-Mathematics-for-Data-Science/Lec-4.1(Descriptive-Statistics).ipynb)





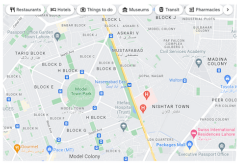


ML is the application of AI that gives machines the ability to learn without being explicitly programmed

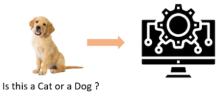


Face Detection

Forecasting



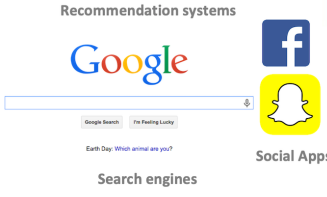
Navigation



Classification

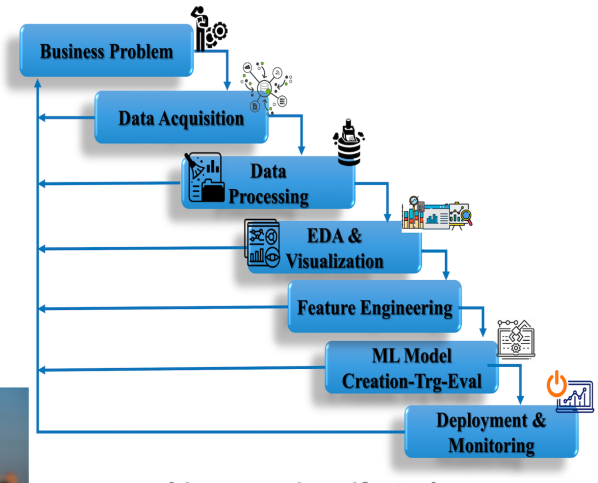
Recommendation systems

Predictions



Search engines

Social Apps



Machine Learning Life Cycle

In [ ]:

```
1
```

In [ ]:

```
1
```

Learning agenda of this notebook

- Overview of Data Pre-Processing and Feature Engineering
  - What is Feature Scaling?

    - Why do we need to Scale our Data?
    - How to perform Feature Scaling?
- Hello Scaling Techniques (`MaxAbsScalar` , `MinMaxScalar` , `StandardScalar` , `RobustScalar`)
    - Using NumPy
    - Using Scikit-Learn Transformers
    - Visualizing data before and after scaling
    - Checking distribution of data before and after scaling
- Perform Scaling on a Real Dataset for ML Task

1. Overview of Data Pre-Processing and Feature Engineering

- `Data Preprocessing` involves actions that we need to perform on the dataset in order to make it ready to be fed to the machine learning model.
- `Feature Engineering` is the process of using domain knowledge to extract features from raw data via data mining techniques.

City	Size	Covered Area	No of bedrooms	Trees near by	No of bathrooms	Schools near by	Construction Date	Price
Lahore	2000	3500	3	1	3	1	25/10/2001	20.5 M
Karachi	2600	3000	2	0	4	1	16/05/1990	18 M
Islamabad	1800	2000	3	1	3	2	25/11/1995	20 M
Shaikhupura	1600	2600	1	2	NaN	0	08/06/2020	5 M
Lahore	2600	2000	3	3	1	1	03/09/2016	4 M
Karachi	3000	1000	2	2	1	NaN	19/01/1980	6 M
Islamabad	2000	3600	44	4	3	3	21/07/1999	30 M
Lahore	1000	2000	3	NaN	1	2	12/04/2015	10 M

- Pre-processing package of sklearn provides a bundle of utility functions and transformer classes for data preprocessing (will cover later).
    - **Detecting and handling outliers**
        - Univariate (Z-Score, IQR, Percentiles)
        - Multivariate Analysis (Depth-based, Distance-based, Density-based methods)
        - Trimming, Capping/Winsorization, Discritization
    - **Missing values Imputation**
        - Univariate Imputation (Panda's `fillna()` method, Sklearn's `SimpleImputer()` transformer)
        - Multivariate Imputation (Sklearn's `IterativeImputer()` and `KNNImputer()` transformers)
    - **Encoding Categorical Features**
        - Encode Nominal i/p features using Pandas `get_dummies()` and Scikit-Learn's `OneHotEncoder()`
        - Encode Ordinal i/p features using Scikit-Learn's `OrdinalEncoder()`
        - Encode Ordinal o/p label using Scikit-Learn's `LabelEncoder()`
    - **Feature Scaling**
        - Use numPy to perform maxabs, minmax, standard and robust scaling
        - Use Sklearn's `MaxAbsScalar` , `MinMaxScalar` , `StandardScalar` , `RobustScalar` transformers
    - **Extracting Information**
        - Use Sklearn's `CountVectorizer` , `DictVectorizer` , `TfidfVectorizer` , and `TfidfTransformer`
    - **Combining Information**
        - Use `FeatureUnion` , `Pipeline` , `PCA`

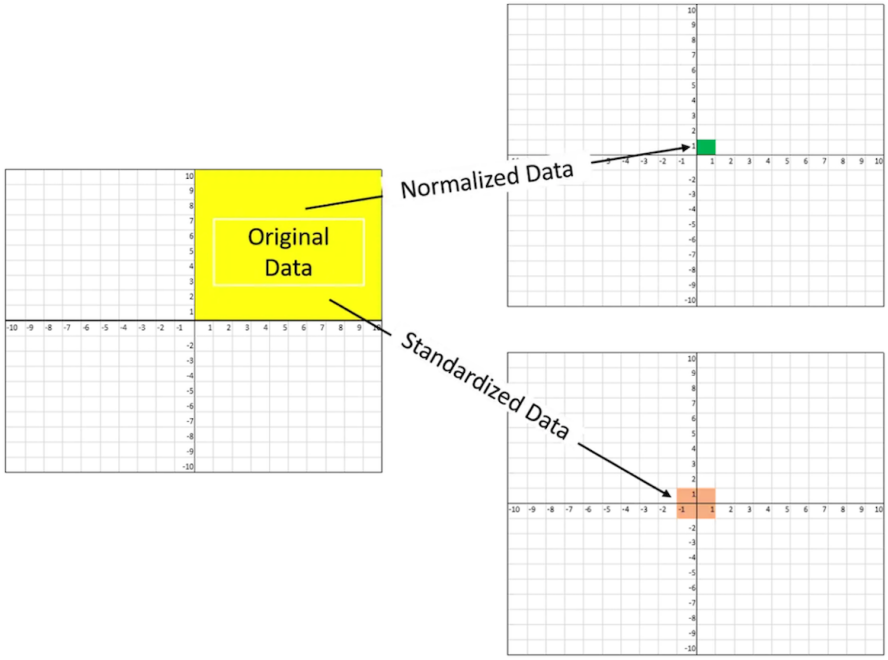
1. Overview of Feature Scaling (What? Why? and How?)

a. What is Feature Scaling?

> **Feature scaling is a process of transforming numeric columns to a common scale**

Original Dataset

	Age	Salary
Person1	44	73000
Person2	27	47000
Person3	30	53000
Person4	38	62000
Person5	40	57000
Person6	35	53000
Person7	48	78000

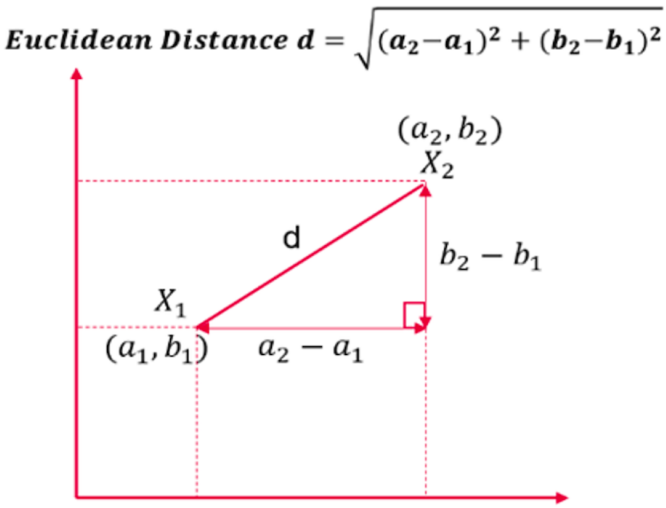


- **Normalization** is rescaling of the data from original range, so that all values are within the new range of 0 and 1.
- **Standardization** is rescaling of the data from original range, so that all values are centered around mean with a standard deviation of 1.
- `Note:`
  - After scaling the distribution of data does not change.
  - Use standardization if your data is normally distributed, otherwise, use normalization
  - Standardization is more robust to outliers.

b. Why do we need to Scale our Data?

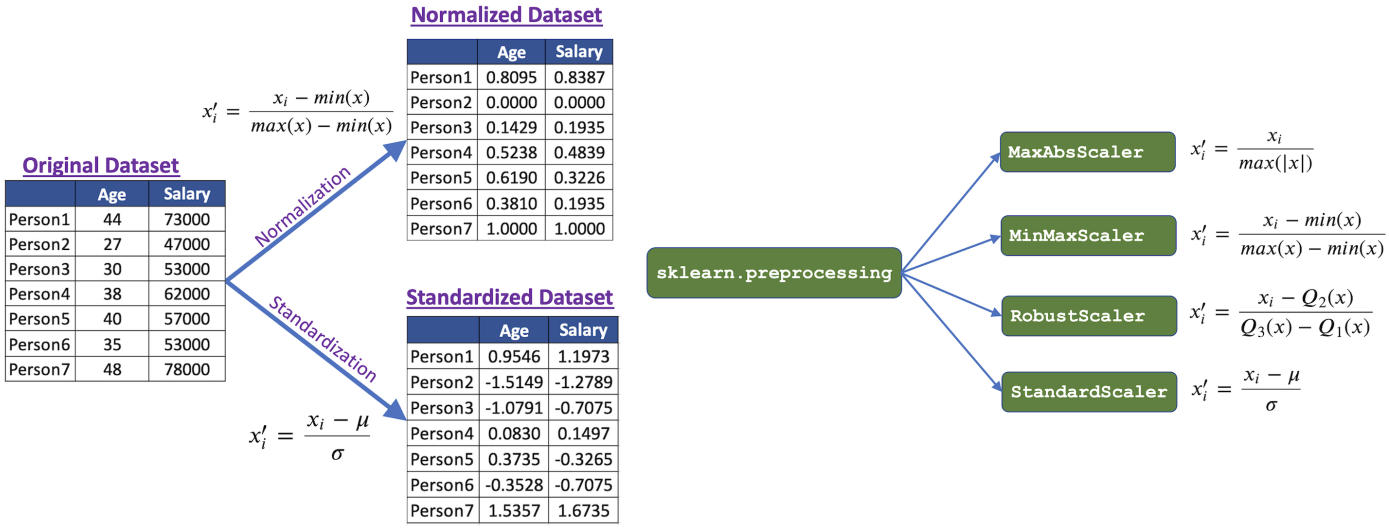
> **Many machine learning algorithms perform better or converge faster, when features are on a relatively similar scale and close to normally distributed**

	Age	Salary
Person1	44	73000
Person2	27	47000
Person3	30	53000
Person4	38	62000
Person5	40	57000
Person6	35	53000
Person7	48	78000



Eculidiean Distance (Person1, Person2): $\sqrt{(27-44)^2 + (47000-73000)^2} = 26000$

c. How to perform Feature Scaling?



	Age	Salary
Person1	44	73000
Person2	27	47000
Person3	30	53000
Person4	38	62000
Person5	40	57000
Person6	35	53000
Person7	48	78000

	Age	Salary
Person1	0.8095	0.8387
Person2	0.0000	0.0000
Person3	0.1429	0.1935
Person4	0.5238	0.4839
Person5	0.6190	0.3226
Person6	0.3810	0.1935
Person7	1.0000	1.0000

	Age	Salary
Person1	0.9546	1.1973
Person2	-1.5149	-1.2789
Person3	-1.0791	-0.7075
Person4	0.0830	0.1497
Person5	0.3735	-0.3265
Person6	-0.3528	-0.7075
Person7	1.5357	1.6735

2. Hello World on Scaling Techniques

In [1]:

```
1 import pandas as pd
2 df = pd.DataFrame({'f1': [15, 18, 12, 10],
3                    'f2': [1, 3, 2, 5]})
4 df
```

Out[1]:

	f1	f2
0	15	1
1	18	3
2	12	2
3	10	5

In [2]:

```
1 df.describe()
```

Out[2]:

	f1	f2
count	4.00	4.000000
mean	13.75	2.750000
std	3.50	1.707825
min	10.00	1.000000
25%	11.50	1.750000
50%	13.50	2.500000
75%	15.75	3.500000
max	18.00	5.000000

- Scale feature columns using NumPy
- Scale feature columns using sklearn transformers
- Visualize data before and after scaling
- Check the distribution of data before and after scaling

a. `MaxAbsScalar`

- MaxAbsScalar divide each data point by the absolute maximum, so that the maximal absolute value of each feature in the training set is 1.0.
- It does not shift/center the data and thus does not destroy any sparsity.
- On positive-only data, this Scaler behaves similarly to Min Max Scaler and, therefore, also suffers from the presence of significant outliers.

$$x_i' = \frac{x_i}{max(|x|)}$$

Using NumPy

In [3]:

```
1 df
```

Out[3]:

	f1	f2
0	15	1
1	18	3
2	12	2
3	10	5

In [4]:

```
import numpy as np
f1_maxabs = [x/np.max(abs(df.f1)) for x in df.f1]
f2_maxabs = [x/abs(np.max(df.f2)) for x in df.f2]
df1 = pd.DataFrame({'f1': f1_maxabs, 'f2': f2_maxabs})
df1
```

Out[4]:

	f1	f2
0	0.833333	0.2
1	1.000000	0.6
2	0.666667	0.4
3	0.555556	1.0

Using `MaxAbsScaler` Transformer

In [5]:

```
from sklearn.preprocessing import MaxAbsScaler
scaler = MaxAbsScaler()
arr = scaler.fit_transform(df)
df2 = pd.DataFrame(arr, columns=['f1','f2'])
df2
```

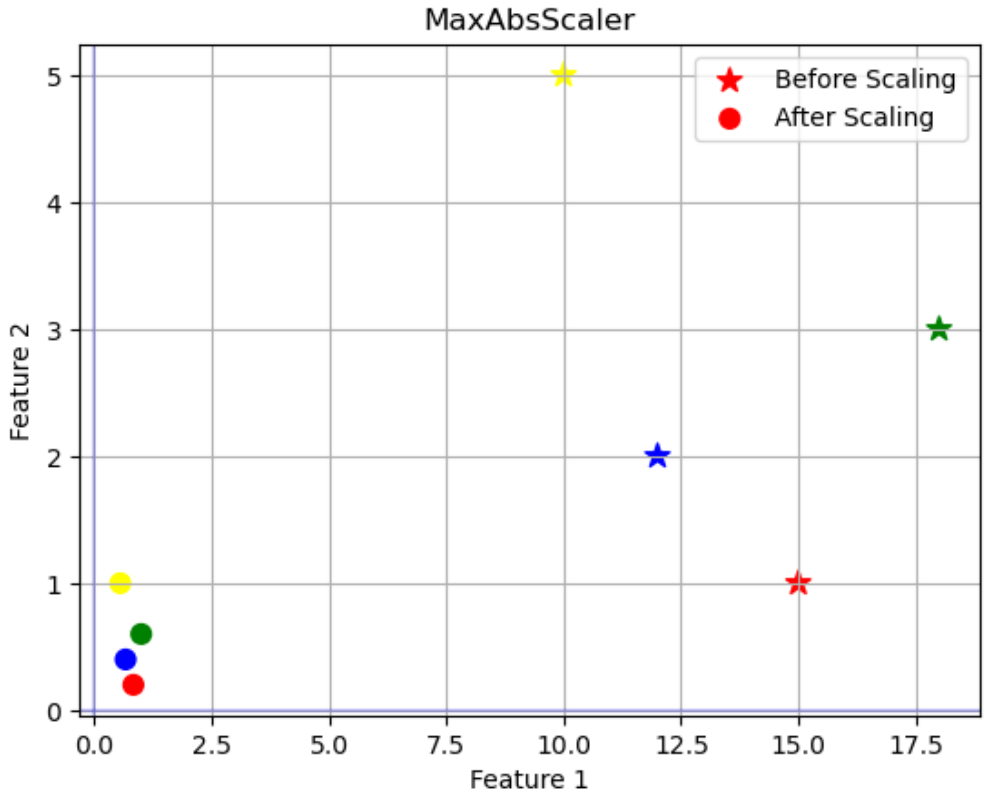
Out[5]:

	f1	f2
0	0.833333	0.2
1	1.000000	0.6
2	0.666667	0.4
3	0.555556	1.0

Visualizing Data Before and After Scaling

In [6]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
colors = ['red','green','blue','yellow']
ax.scatter(x=df.f1, y=df.f2, label="Before Scaling", color=colors, marker = '*',s=100)
ax.scatter(x=df2.f1, y=df2.f2, label="After Scaling", color=colors, marker = 'o',s=60)
ax.set_xlabel("Feature 1")
ax.set_ylabel("Feature 2")
plt.title("MaxAbsScaler")
plt.legend(loc='best')
plt.axhline(0, color='blue',alpha=0.2)
plt.axvline(0, color='blue',alpha=0.2);
plt.grid(True)
```



b. `MinMaxScalar`

- The `MinMaxScalar` scales and translates the feature values such that they fall in the range of [0,1] . If there are negative values, it shrinks the data within the range of [-1,1].

$$x_i' = \frac{x_i - min(x)}{max(x) - min(x)}$$

- When the value of x_i is the minimum value in the column, the numerator will be 0, and hence x_i' will be 0
- On the other hand, when the value of x_i is the maximum value in the column, the numerator is equal to the denominator and thus the value of x_i' will be 1
- If the value of x_i is between the minimum and the maximum value, then the value of x_i' will be between 0 and 1

> - This Scaler responds well if the standard deviation is small and when a distribution is not Gaussian. This Scaler is sensitive to outliers.

In [7]:

```
df
```

Out[7]:

	f1	f2
0	15	1
1	18	3
2	12	2
3	10	5

Using NumPy

In [8]:

```
f1_minmax = [(x-np.min(df.f1))/(np.max(df.f1) - np.min(df.f1)) for x in df.f1]
f2_minmax = [(x-np.min(df.f2))/(np.max(df.f2) - np.min(df.f2)) for x in df.f2]
df1 = pd.DataFrame({'f1': f1_minmax, 'f2': f2_minmax})
df1
```

Out[8]:

	f1	f2
0	0.625	0.00
1	1.000	0.50
2	0.250	0.25
3	0.000	1.00

Using `MinMaxScaler` Transformer

In [9]:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
arr = scaler.fit_transform(df)
df2 = pd.DataFrame(arr, columns=['f1','f2'])
df2
```

Out[9]:

	f1	f2
0	0.625	0.00
1	1.000	0.50
2	0.250	0.25
3	0.000	1.00

Visualize Data Before and After Scaling

In [ ]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
colors = ['red','green','blue','yellow']
ax.scatter(x=df.f1, y=df.f2, label="Before Scaling", color=colors, marker = '*',s=100)
ax.scatter(x=df2.f1, y=df2.f2, label="After Scaling", color=colors, marker = 'o',s=60)
ax.set_xlabel("Feature 1")
ax.set_ylabel("Feature 2")
plt.title("MinMaxScaler")
plt.legend(loc='best')
plt.axhline(0, color='blue',alpha=0.2)
plt.axvline(0, color='blue',alpha=0.2);
plt.grid(True)
```

c. `StandardScalar`

- StandardScaler is the industry's go-to algorithm 🙂
- It rescales the data from original range, so that all values are centered around the mean of zero with a standard deviation of one.
- StandardScalar assumes that you are working with normally distributed data and the outliers are not too crazy

$$Z_i = x_i' = \frac{x_i - \mu}{\sigma}$$

- `Note:` The values are not restricted to a particular range, although majority will fall within 1σ around mean

In [ ]:

```
df
```

Using NumPy

In [10]:

```
f1_std = [(x-np.mean(df.f1))/(np.std(df.f1)) for x in df.f1]
f2_std = [(x-np.mean(df.f2))/(np.std(df.f2)) for x in df.f2]
df1 = pd.DataFrame({'f1': f1_std, 'f2': f2_std})
df1
```

Out[10]:

	f1	f2
0	0.412393	-1.183216
1	1.402136	0.169031
2	-0.577350	-0.507093
3	-1.237179	1.521278

Using `StandardScaler` Transformer

In [11]:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
arr = scaler.fit_transform(df)
df2 = pd.DataFrame(arr, columns=['f1','f2'])
df2
```

Out[11]:

	f1	f2
0	0.412393	-1.183216
1	1.402136	0.169031
2	-0.577350	-0.507093
3	-1.237179	1.521278

In [12]:

```
df2.f1.mean()
```

Out[12]:

```
5.551115123125783e-17
```

Visualize Data Before and After Scaling

In [ ]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
colors = ['red','green','blue','yellow']
ax.scatter(x=df.f1, y=df.f2, label="Before Scaling", color=colors, marker = '*',s=100)
ax.scatter(x=df2.f1, y=df2.f2, label="After Scaling", color=colors, marker = 'o',s=60)
ax.set_xlabel("Feature 1")
ax.set_ylabel("Feature 2")
plt.title("StandardScaler")
plt.legend(loc='best')
plt.axhline(0, color='blue',alpha=0.2)
plt.axvline(0, color='blue',alpha=0.2);
plt.grid(True)
```

d. `RobustScalar`

- As the name suggests, this Scaler is robust to outliers.
- Outliers can skew a probability distribution and make data scaling using standardization difficult as the calculated **mean and standard deviation will be skewed by the presence of the outliers.**
- Instead of subtracting the mean, you subtract data values from the median (50th percentile), and instead of dividing by the standard deviation, you divide by Inter Quartile Range (IQR).
- The formula to scale using Robust Scalar using quartiles:

$$x_i' = \frac{x_i - Q_2(x)}{Q_3(x) - Q_1(x)}$$

- The formula to scale using Robust Scalar using percentiles:

$$x_i' = \frac{x_i - Median}{P_{75} - P_{25}}$$

- The resulting variable will have a zero mean and median and a standard deviation of 1, although not skewed by outliers and the outliers are still present with the same relative relationships to other values.

In [ ]:

```
df
```

Using NumPy

In [ ]:

```
f1_robust = [(x-df.f1.quantile(0.50))/(df.f1.quantile(0.75) - df.f1.quantile(0.25)) for x
f2_robust = [(x-df.f2.quantile(0.50))/(df.f2.quantile(0.75) - df.f2.quantile(0.25)) for x
df1 = pd.DataFrame({'f1': f1_robust, 'f2': f2_robust})
df1
```

Using `RobustScaler` Transformer

In [ ]:

```
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()
arr = scaler.fit_transform(df)
df2 = pd.DataFrame(arr, columns=['f1','f2'])
df2
```

Visualizing Data Before and After Scaling

In [ ]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
colors = ['red','green','blue','yellow']
ax.scatter(x=df.f1, y=df.f2, label="Before Scaling", color=colors, marker = '*',s=100)
ax.scatter(x=df2.f1, y=df2.f2, label="After Scaling", color=colors, marker = 'o',s=60)
ax.set_xlabel("Feature 1")
ax.set_ylabel("Feature 2")
plt.title("RobustScaler")
plt.legend(loc='best')
plt.axhline(0, color='blue',alpha=0.2)
plt.axvline(0, color='blue',alpha=0.2);
plt.grid(True)
```

e. Comparison

Original

	f1	f2
0	15	1
1	18	3
2	12	2
3	10	5

MaxAbsScalar

	f1	f2
0	0.833333	0.2
1	1.000000	0.6
2	0.666667	0.4
3	0.555556	1.0

MinMaxScalar

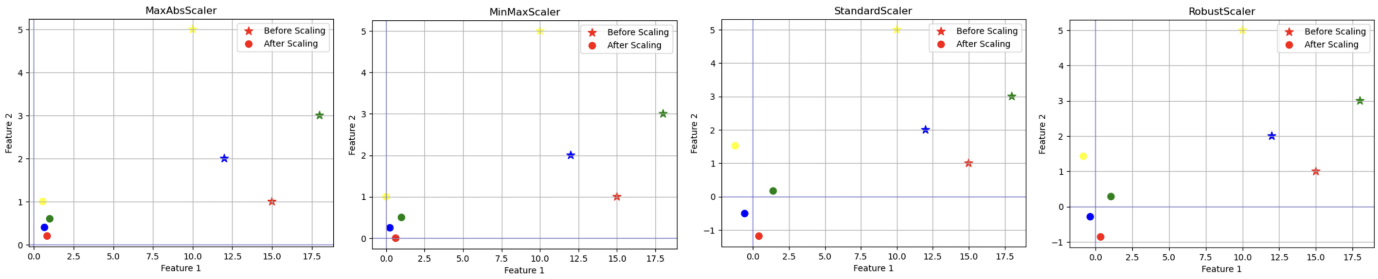
	f1	f2
0	0.625	0.00
1	1.000	0.50
2	0.250	0.25
3	0.000	1.00

StandardScalar

	f1	f2
0	0.412393	-1.183216
1	1.402136	0.169031
2	-0.577350	-0.507093
3	-1.237179	1.521278

RobustScalar

	f1	f2
0	0.352941	-0.857143
1	1.058824	0.285714
2	-0.352941	-0.285714
3	-0.823529	1.428571



f. Effect of Outliers in the Distribution

Import Libraries:

In [13]:

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
matplotlib.style.use('fivethirtyeight')
```

Create synthetic dataset:

In [14]:

```
# Input Feature with lower outliers (1000 values with a mean of )
arr1 = np.concatenate([np.random.normal(loc=20, scale=2, size=1000), np.random.normal(loc
# Input Feature with higher outliers
arr2 = np.concatenate([np.random.normal(loc=40, scale=2, size=1000), np.random.normal(loc
df = pd.DataFrame({'f1': arr1,'f2': arr2})
df
```

Out[14]:

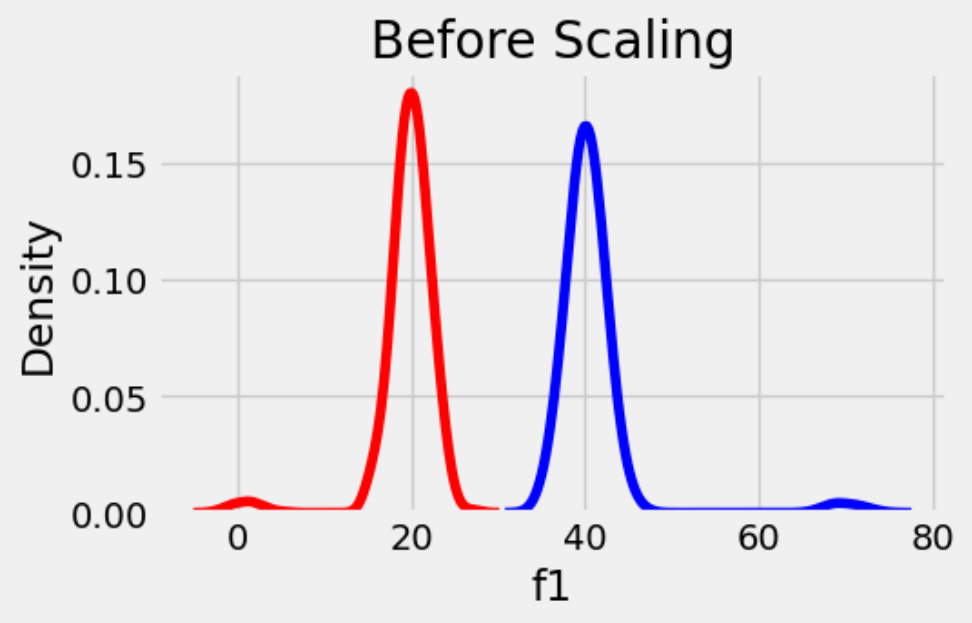
	f1	f2
0	20.775900	38.497291
1	17.945257	40.447157
2	21.883311	42.295975
3	16.165465	38.084696
4	21.381038	38.962157
...	...	...
1020	1.336110	71.103113
1021	-0.013440	69.025217
1022	-0.698553	72.272227
1023	-0.143517	70.729255
1024	0.710971	72.007431

1025 rows × 2 columns

Visualize Distribution of the two feature columns:

In [15]:

```
fig, ax = plt.subplots(figsize =(5, 3))
ax.set_title('Before Scaling')
sns.kdeplot(df['f1'], color ='r')
sns.kdeplot(df['f2'], color ='b')
plt.show()
```



- Observe feature 1 (red) has a mean of 20, while the outliers in feature 1 (red) are around a mean of 1
- Observe feature 2 (blue) has a mean of 40, while the outliers in feature 2 (blue) are around a mean of 70

Perform Scaled Dataframes using MinMaxScaler, StandardScaler, and RobustScaler:

In [16]:

```
scaler = preprocessing.MinMaxScaler()
minmax_df = scaler.fit_transform(df)
minmax_df = pd.DataFrame(minmax_df, columns =['f1', 'f2'])

scaler = preprocessing.StandardScaler()
standard_df = scaler.fit_transform(df)
standard_df = pd.DataFrame(standard_df, columns =['f1', 'f2'])

scaler = preprocessing.RobustScaler()
robust_df = scaler.fit_transform(df)
robust_df = pd.DataFrame(robust_df, columns =['f1', 'f2'])
```

Visualize Distributions before and after Scaling:

In [17]:

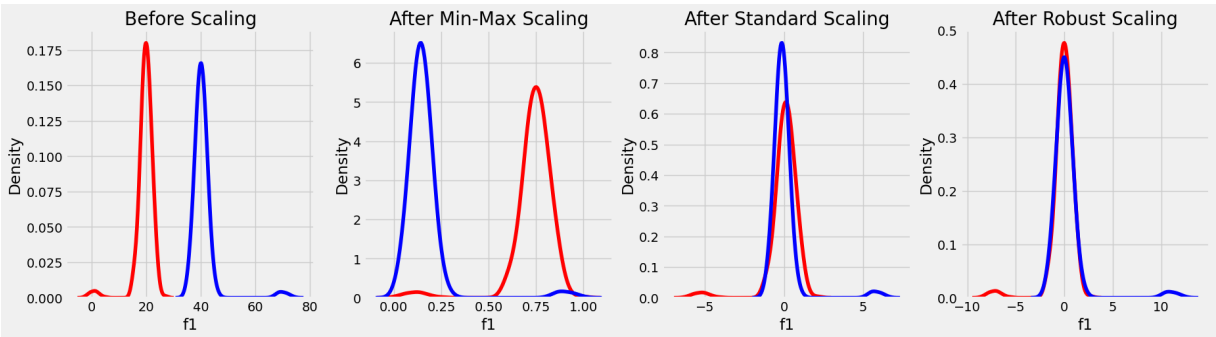
```
fig, (ax1, ax2, ax3, ax4) = plt.subplots(ncols = 4, figsize =(20, 5))
ax1.set_title('Before Scaling')
sns.kdeplot(df['f1'], ax = ax1, color ='r')
sns.kdeplot(df['f2'], ax = ax1, color ='b')

ax2.set_title('After Min-Max Scaling')
sns.kdeplot(minmax_df['f1'], ax = ax2, color ='r')
sns.kdeplot(minmax_df['f2'], ax = ax2, color ='b')

ax3.set_title('After Standard Scaling')
sns.kdeplot(standard_df['f1'], ax = ax3, color ='r')
sns.kdeplot(standard_df['f2'], ax = ax3, color ='b')

ax4.set_title('After Robust Scaling')
sns.kdeplot(robust_df['f1'], ax = ax4, color ='r')
sns.kdeplot(robust_df['f2'], ax = ax4, color ='b')

plt.show()
```



3. Perform Scaling on a Real Dataset for ML Task

- It is always a better approach to train-test-split first and then do the scaling so that there is no data leakage.
- So you `fit_transform()` on the train data, then only transform on the test data.
- Almost all feature engineering like missing value imputation, encoding of categorical data, feature scaling etc should be done after train-test-split.

a. Load Dataset

In [18]:

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
matplotlib.style.use('fivethirtyeight')
df = pd.read_csv('datasets/scaling-data1.csv')
df
```

Out[18]:

	age	salary	target
0	19	19000	0
1	35	20000	0
2	26	43000	0
3	27	57000	0
4	19	76000	0
...	...	...	...
395	46	41000	1
396	51	23000	1
397	50	20000	1
398	36	33000	0
399	49	36000	1

400 rows × 3 columns

In [19]:

```
df.describe()
```

Out[19]:

	age	salary	target
count	400.000000	400.000000	400.000000
mean	37.655000	69742.500000	0.357500
std	10.482877	34096.960282	0.479864
min	18.000000	15000.000000	0.000000
25%	29.750000	43000.000000	0.000000
50%	37.000000	70000.000000	0.000000
75%	46.000000	88000.000000	1.000000
max	60.000000	150000.000000	1.000000

In [20]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   age     400 non-null    int64
 1   salary  400 non-null    int64
 2   target  400 non-null    int64
dtypes: int64(3)
memory usage: 9.5 KB
```

b. Do a Train-Test-Split

In [21]:

```
X = df.drop('target', axis=1)
X
```

Out[21]:

	age	salary
0	19	19000
1	35	20000
2	26	43000
3	27	57000
4	19	76000
...	...	...
395	46	41000
396	51	23000
397	50	20000
398	36	33000
399	49	36000

400 rows × 2 columns

In [22]:

```
y = df['target']
y
```

Out[22]:

```
0      0
1      0
2      0
3      0
4      0
      ..
395    1
396    1
397    1
398    0
399    1
Name: target, Length: 400, dtype: int64
```

In [23]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=54
len(X_train), len(y_train), len(X_test), len(y_test)
```

Out[23]:

```
(320, 320, 80, 80)
```

In [24]:

```
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[24]:

```
((320, 2), (80, 2), (320,), (80,))
```

In [25]:

```
X_train.head()
```

Out[25]:

	age	salary
333	40	65000
273	39	106000
307	47	113000
4	19	76000
292	55	39000

In [26]:

```
y_train.head()
```

Out[26]:

```
333    0
273    1
307    1
4      0
292    1
Name: target, dtype: int64
```

In [27]:

```
X_test.head()
```

Out[27]:

	age	salary
346	53	72000
178	24	23000
251	37	52000
228	40	72000
179	31	34000

In [28]:

```
y_test.head()
```

Out[28]:

```
346    1
178    0
251    0
228    0
179    0
Name: target, dtype: int64
```

c. Apply `StandardScaler` to `age` and `salary` Columns

In [29]:

```
from sklearn.preprocessing import StandardScaler

ss = StandardScaler()

# fit the scaler to the train set, it will learn the parameters
ss.fit(X_train[['age','salary']])

# transform train and test sets
X_train_scaled = ss.transform(X_train[['age','salary']])
X_test_scaled = ss.transform(X_test[['age','salary']])

X_train_scaled.shape, X_test_scaled.shape
```

Out[29]:

```
((320, 2), (80, 2))
```

In [30]:

```
from sklearn.preprocessing import StandardScaler, RobustScaler, MinMaxScaler, MaxAbsScale
```

In [31]:

```
# convert the two numPy arrays to Pandas dataframes
X_train_scaled = pd.DataFrame(X_train_scaled, columns=X_train.columns)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X_test.columns)
```

In [32]:

```
1 X_train_scaled
```

Out[32]:

	age	salary
0	0.226798	-0.160423
1	0.132176	1.040916
2	0.889155	1.246022
3	-1.760273	0.161888
4	1.646135	-0.922247
...	...	...
315	0.699910	-1.420363
316	2.024624	0.161888
317	1.078400	-0.922247
318	0.321421	-0.306927
319	-0.057069	0.015383

320 rows × 2 columns

In [33]:

```
1 X_test_scaled
```

Out[33]:

	age	salary
0	1.456890	0.044684
1	-1.287161	-1.391062
2	-0.057069	-0.541335
3	0.226798	0.044684
4	-0.624804	-1.068751
...	...	...
75	0.226798	-0.394830
76	-0.246314	0.132587
77	0.037553	-0.599937
78	0.416043	-0.482733
79	-0.057069	0.103286

80 rows × 2 columns

In [34]:

```
1 np.round(X_train_scaled.describe(), 2) #rounding the values for better visualization
```

Out[34]:

	age	salary
count	320.00	320.00
mean	0.00	-0.00
std	1.00	1.00
min	-1.85	-1.63
25%	-0.81	-0.78
50%	-0.06	-0.03
75%	0.79	0.51
max	2.12	2.33

In [35]:

```
1 np.round(X_test_scaled.describe(), 2) #rounding the values for better visualization
```

Out[35]:

	age	salary
count	80.00	80.00
mean	0.02	-0.11
std	0.96	0.99
min	-1.85	-1.63
25%	-0.72	-0.95
50%	0.04	-0.01
75%	0.42	0.32
max	2.12	2.30

d. Visualize Distribution Before and After Scaling

In [36]:

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols = 2, figsize =(25, 15))
# age before scaling
ax1.set_title('Age Before Scaling')
sns.kdeplot(X_train['age'], ax=ax1, color ='red')

# age after scaling
ax2.set_title('Age After Standard Scaling')
sns.kdeplot(X_train_scaled['age'], ax=ax2, color ='red')

# salary before scaling
ax3.set_title('Salary Before Scaling')
sns.kdeplot(X_train['salary'], ax=ax3, color ='blue')

# salary after scaling
ax4.set_title('Salary After Standard Scaling')
sns.kdeplot(X_train_scaled['salary'], ax=ax4, color ='blue')
plt.show()
```