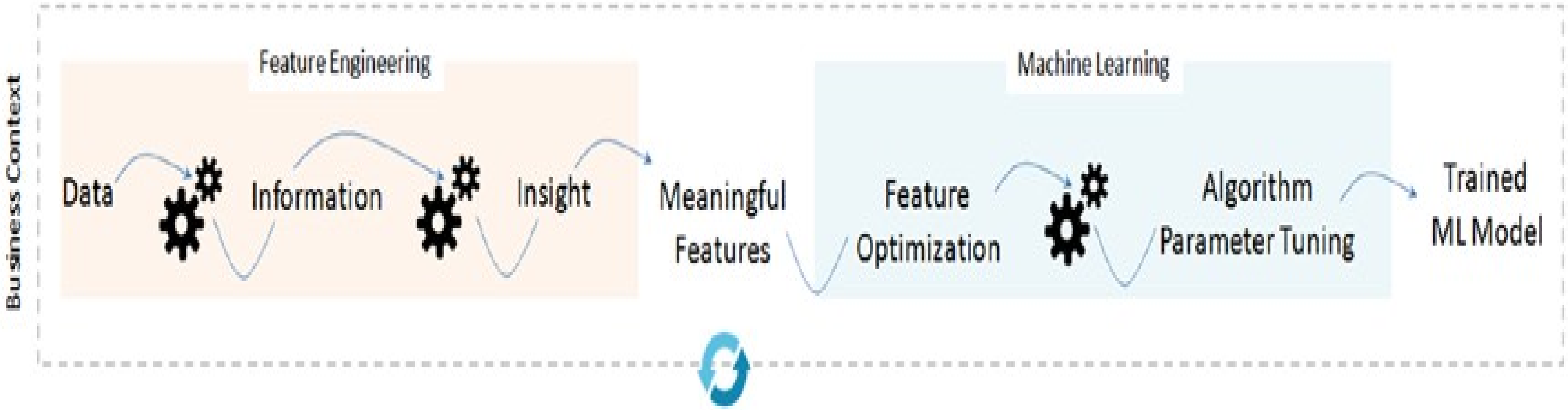
FUNDAMENTALS of MACHINE LEARNING using PYTHON and SCIKIT-LEARN

Feature Engineering

The output or the prediction quality of any ML algorithm predominantly depends on the quality of input being passed. **The process of creating appropriate data features by applying the business context is called feature engineering**, and it is one of the most important aspects of building an efficient ML system. The business context here means the expression of the business problem that we are trying to address, why we are trying to solve it, and what is the expected outcome. So let’s understand the fundamentals of feature engineering before proceeding to different types of ML algorithms.



Dealing with Missing Data

Missing data can mislead or create problems for analyzing the data. In order to avoid any such issues, you need to impute missing data. There are four most commonly used techniques for data imputation:

•**Delete:** You could simply delete the rows containing missing values. This technique is more suitable and effective when a number of missing values row count is insignificant (say <5%) compare with the overall record count. You can achieve this using Panda's **dropna()** function.

•**Replace with the summary:** This is probably the most commonly used imputation technique. Summarization here is the mean, mode, or median for a respective column. For continuous or quantitative variables, either mean/average or mode or median value of the respective column can be used to replace the missing values. Whereas for categorical or qualitative variables, the mode (most frequent) summation technique works better. You can achieve this using Panda's **fillna()** function.

•**Random replace:** You can also replace the missing values with a randomly picked value from the respective column. This technique would be appropriate where missing values row count is insignificant.

•**Use a predictive model:** This is an advanced technique. Here you can train a regression model for continuous variables and a classification model for categorical variables with the available data and use the model to predict the missing values.

Handling Categorical Data

Some common methods of handling categorical data, based on their number of levels:

Create a dummy variable: This is a Boolean variable that indicates the presence of a category with the value 1 and 0 for absence. You should create k-1 dummy variables, where k is the number of levels (indexing in Python starts from 0).

Code_1:

```
import pandas as pd
from patsy import dmatrices
df = pd.DataFrame({'A': ['high', 'medium', 'low'], 'B': [10,20,30]}, index=[0, 1, 2])
print(df)
```

Output:

```
          A                   B
0         high                10
1         medium              20
2         low                 30
```

Rescaling a Feature

Rescaling is a common preprocessing task in machine learning. Many of the algorithms described later will assume all features are on the same scale, typically 0 to 1 or –1 to 1. There are a number of rescaling techniques, but one of the simplest is called ***min-max scaling***. This technique uses the minimum and maximum values of a feature to rescale values to within a range. Specifically, min-max calculates:

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

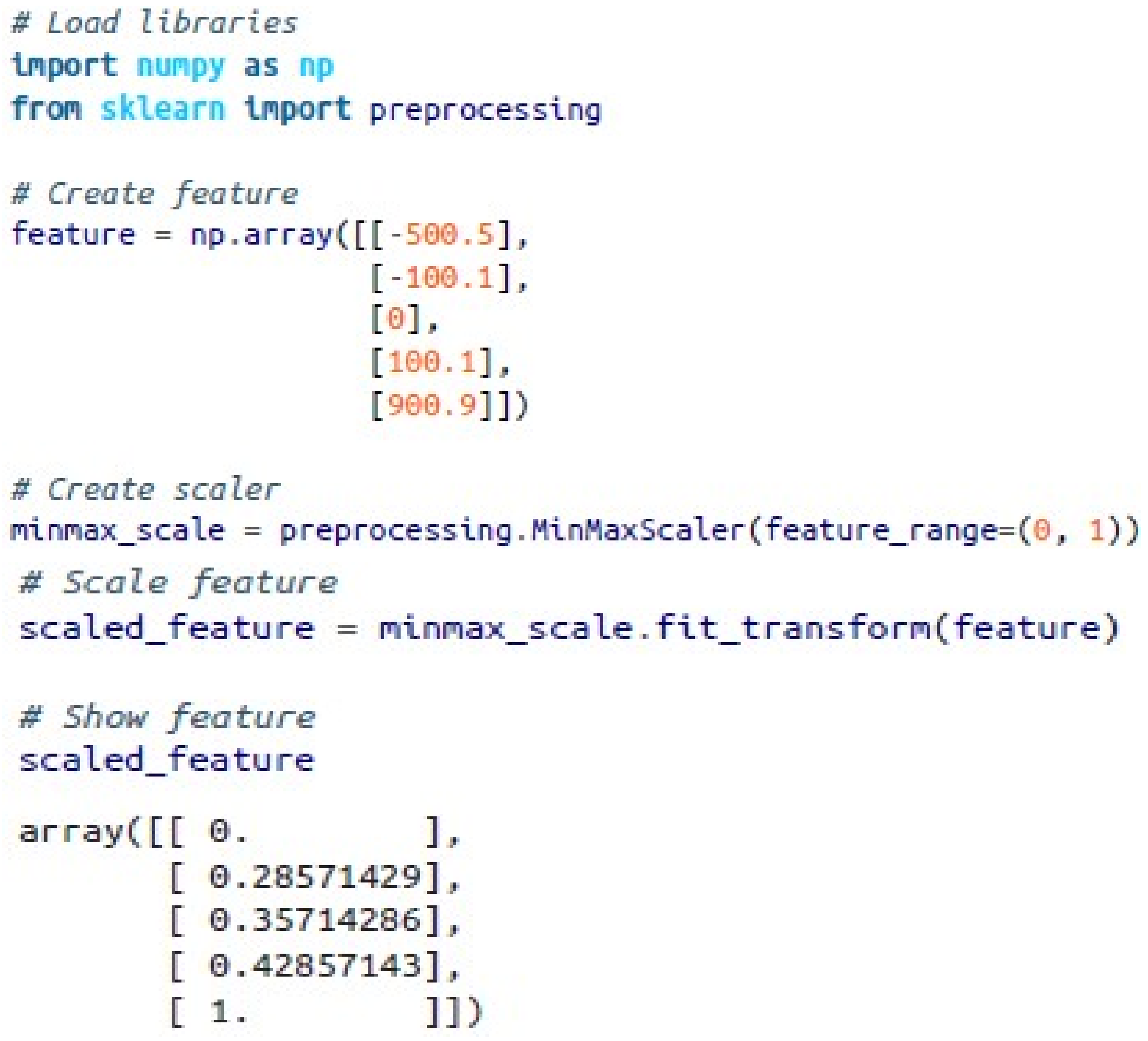
Where,

x is the feature vector,

x_i is an individual element of feature *x*, and

x'_i is the rescaled element.

In our example, we can see from the outputted array that the feature has been successfully rescaled to between 0 and 1:



```
# Load libraries
import numpy as np
from sklearn import preprocessing

# Create feature
feature = np.array([[-500.5],
                    [-100.1],
                    [0],
                    [100.1],
                    [900.9]])

# Create scaler
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))
# Scale feature
scaled_feature = minmax_scale.fit_transform(feature)

# Show feature
scaled_feature

array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

Standardizing a Feature

A common alternative to min-max scaling is rescaling of features to be approximately standard normally distributed. To achieve this, we use standardization to tranform the data such that it has a mean, $\bar{x}$ or 0 and a standard deviation σ, of 1. Specifically, each element in the feature is transformed so that:

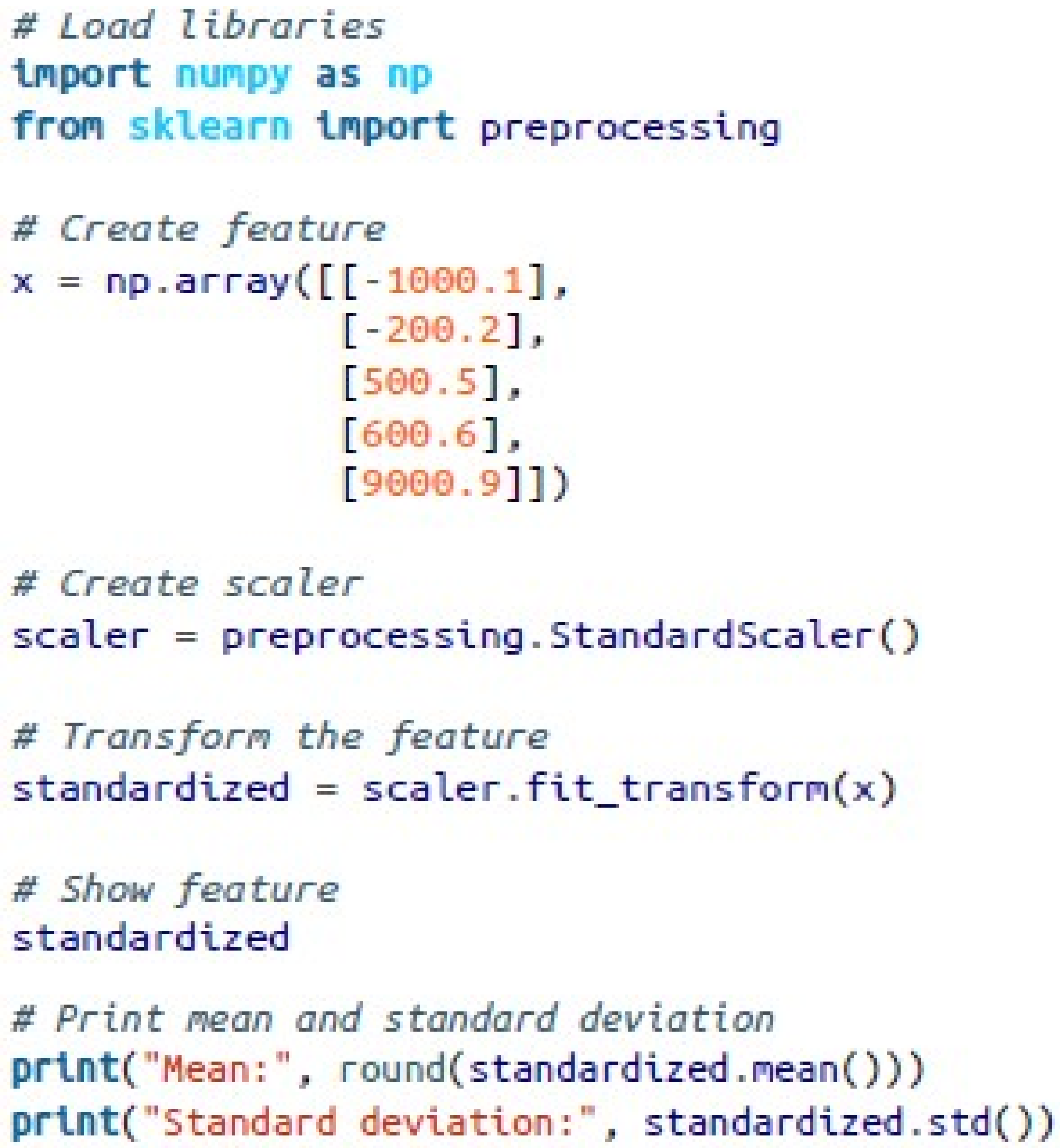
$$x_i' = (xi - \bar{x})/\sigma$$

Where x_i' is our standardized form of x_i. The transformed feature represents the number of standard deviations in the original value is away from the feature's mean value (also called a *z-score* in statistics).

The principal component analysis (PCA) often works better using standardization, while min-max scaling is often recommended for neural networks.

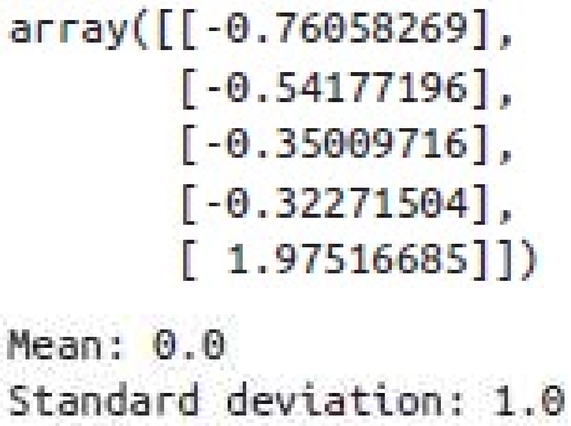
scikit-learn's StandardScaler transforms a feature to have a mean of 0 and a standard deviation of 1.

Standardizing a Feature…contd.



```python
# Load libraries
import numpy as np
from sklearn import preprocessing

# Create feature
x = np.array([[-1000.1],
              [-200.2],
              [500.5],
              [600.6],
              [9000.9]])

# Create scaler
scaler = preprocessing.StandardScaler()

# Transform the feature
standardized = scaler.fit_transform(x)

# Show feature
standardized

# Print mean and standard deviation
print("Mean:", round(standardized.mean()))
print("Standard deviation:", standardized.std())
```

OUTPUT:



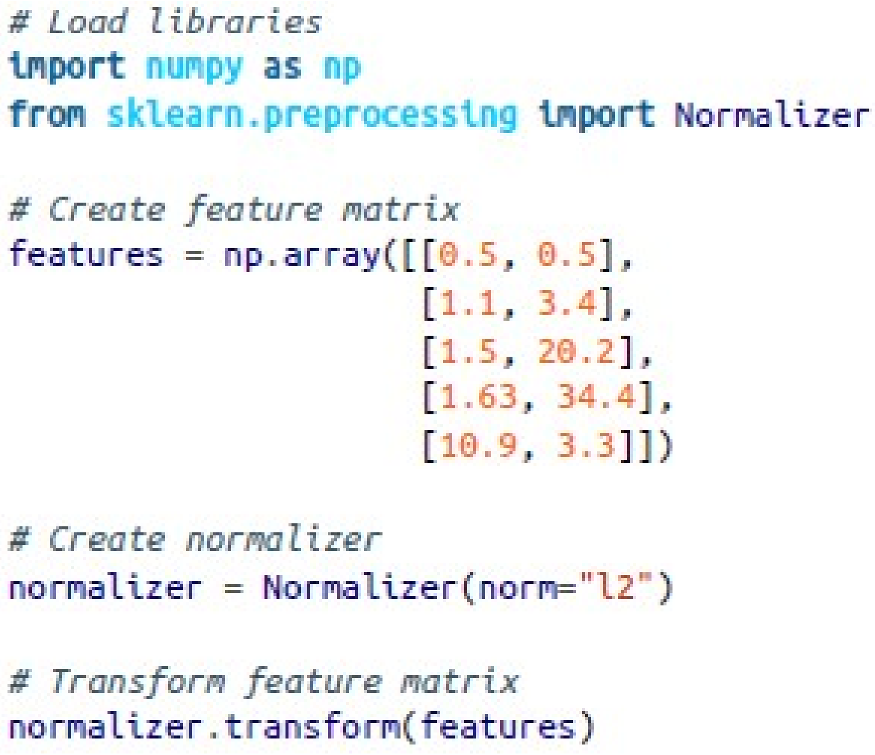
```
array([[-0.76058269],
       [-0.54177196],
       [-0.35009716],
       [-0.32271504],
       [ 1.97516685]])

Mean: 0.0
Standard deviation: 1.0
```

Normalizing Data

Scikit-learn's **Normalizer** rescales the values on individual observations to have unit norm (the sum of their lengths is 1). This type of rescaling is often used when we have many equivalent features (e.g., text classification when every word or *n-word group is a feature)*. Normalizer provides Euclidean norm (often called L2) being the default argument:

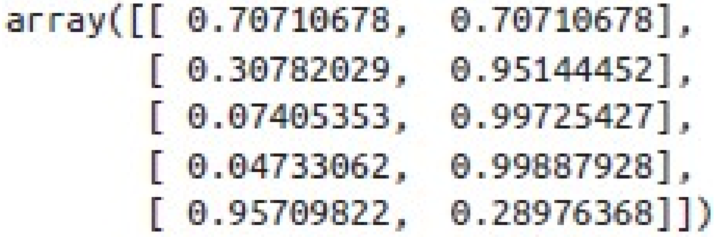
$$\| x \|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

where x is an individual observation and x_n is that observation’s value for the nth feature.



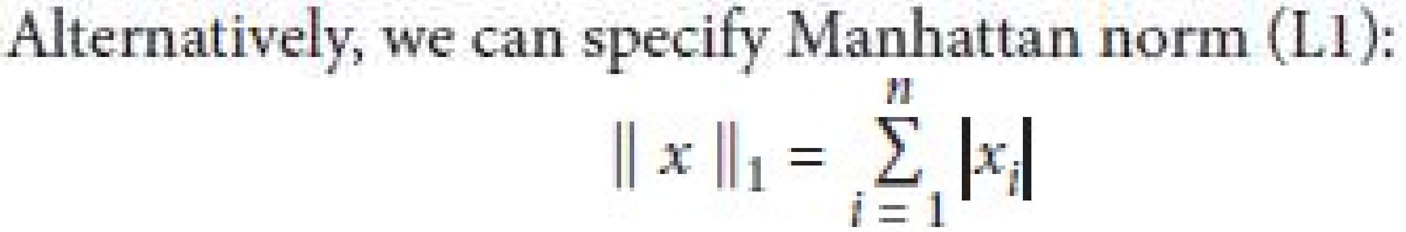
```
# Load libraries
import numpy as np
from sklearn.preprocessing import Normalizer

# Create feature matrix
features = np.array([[0.5, 0.5],
                     [1.1, 3.4],
                     [1.5, 20.2],
                     [1.63, 34.4],
                     [10.9, 3.3]])

# Create normalizer
normalizer = Normalizer(norm="l2")

# Transform feature matrix
normalizer.transform(features)
```

OUTPUT:



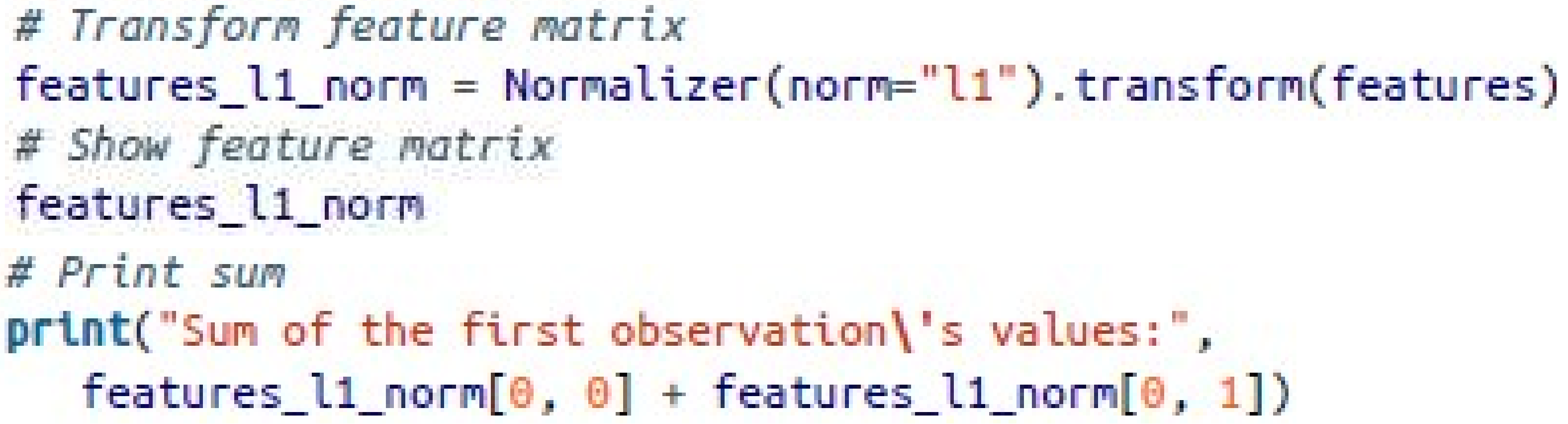
```
array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

Normalizing Data…contd.



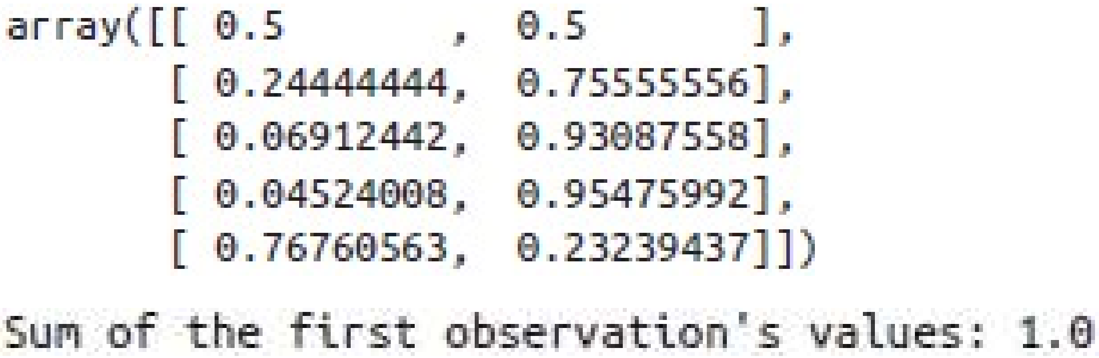
Alternatively, we can specify Manhattan norm (L1):

$$\| x \|_1 = \sum_{i=1}^{n} |x_i|$$



```
# Transform feature matrix
features_l1_norm = Normalizer(norm="l1").transform(features)
# Show feature matrix
features_l1_norm
# Print sum
print("Sum of the first observation\'s values:",
   features_l1_norm[0, 0] + features_l1_norm[0, 1])
```

OUTPUT:



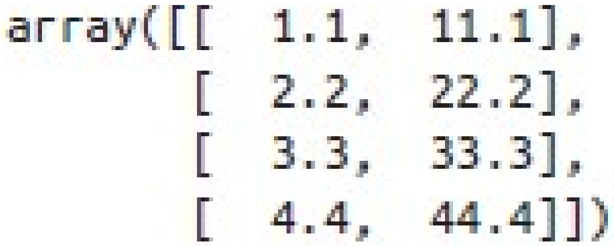
```
array([[ 0.5       ,  0.5       ],
       [ 0.24444444,  0.75555556],
       [ 0.06912442,  0.93087558],
       [ 0.04524008,  0.95475992],
       [ 0.76760563,  0.23239437]])

Sum of the first observation's values: 1.0
```

Deleting Observations with Missing Values

Deleting observations with missing values is easy with a clever line of NumPy:

```
# Load library
import numpy as np
# Create feature matrix
features = np.array([[1.1, 11.1],
[2.2, 22.2],
[3.3, 33.3],
[4.4, 44.4],
[np.nan, 55]])
# Keep only observations that are not (denoted by ~) missing
features[~np.isnan(features).any(axis=1)]
```

OUTPUT:



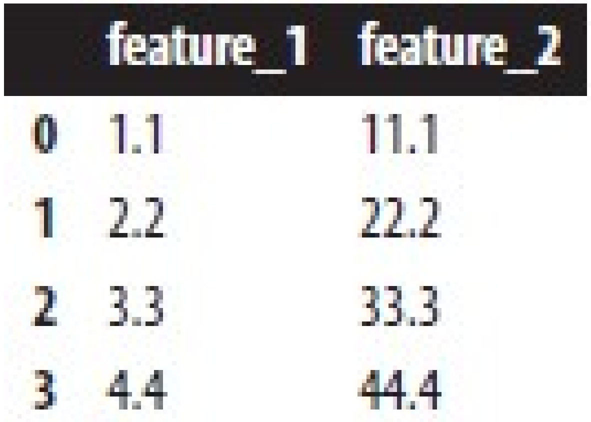
```
array([[  1.1,  11.1],
       [  2.2,  22.2],
       [  3.3,  33.3],
       [  4.4,  44.4]])
```

Deleting Observations with Missing Values…contd.

Alternatively, we can drop missing observations using pandas:

```
# Load library
import pandas as pd
# Load data
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])
# Remove observations with missing values
dataframe.dropna()
```

OUTPUT:



	feature_1	feature_2
0	1.1	11.1
1	2.2	22.2
2	3.3	33.3
3	4.4	44.4

Deleting Observations with Missing Values…contd.

Most machine learning algorithms cannot handle any missing values in the target and feature arrays. For this reason, we cannot ignore missing values in our data and must address the issue during preprocessing.

The simplest solution is to delete every observation that contains one or more missing values, a task quickly and easily accomplished using NumPy or pandas.

Depending on the cause of the missing values, deleting observations can introduce bias into our data.

There are three types of missing data:

1. **Missing Completely At Random (MCAR)**

The probability that a value is missing is independent of everything. For example, a survey respondent rolls a die before answering a question: if he/she rolls a six, he/she skips that question.

Deleting Observations with Missing Values…contd.

2. Missing At Random (MAR)

The probability that a value is missing is not completely random, but depends on the information captured in other features. For example, a survey asks about gender identity and annual salary and women are more likely to skip the salary question; however, their non-response depends only on information we have captured in our gender identity feature.

3. Missing Not At Random (MNAR)

The probability that a value is missing is not random and depends on information not captured in our features. For example, a survey asks about gender identity and women are more likely to skip the salary question, and we do not have a gender identity feature in our data.

It is sometimes acceptable to **delete observations if they are MCAR or MAR**. However, **if the value is MNAR, the fact that a value is missing is itself information**. Deleting MNAR observations can inject bias into our data because we are removing observations produced by some unobserved systematic effect.

Univariate Analysis

Individual variables are analyzed in isolation to get a better understanding of them. Pandas provides a describe function to create summary statistics in tabular format for all variables.These statistics are very useful for the numerical type of variables, to understand any quality issues such as missing value and presence of outliers.

```
from sklearn import datasets
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
iris = datasets.load_iris()
# Let's convert to dataframe
iris = pd.DataFrame(data= np.c_[iris['data'], iris['target']],
            columns= iris['feature_names'] + ['species'])
# replace the values with class labels
iris.species = np.where(iris.species == 0.0, 'setosa', np.where(iris.
            species==1.0,'versicolor', 'virginica'))
# let's remove spaces from column name
iris.columns = iris.columns.str.replace (' ', ")
iris.describe()
```

Multivariate Analysis

In multivariate analysis, we try to establish a sense of relationship of all variables with one other. Let’s determine the mean of each feature by species type:

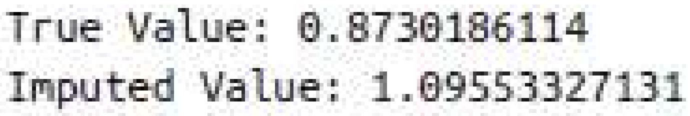
```
# print the mean for each column by species
iris.groupby(by = "species").mean()
# plot for mean of each feature for each label class
iris.groupby(by = "species").mean().plot(kind="bar")
plt.title('Class vs Measurements')
plt.ylabel('mean measurement(cm)')
plt.xticks(rotation=0) # manage the xticks rotation
plt.grid(True)
# Use bbox_to_anchor option to place the legend outside plot area to be tidy
plt.legend(loc="upper left", bbox_to_anchor=(1,1))
```

Imputing Missing Values

If you have a small amount of data, **predict the missing values using k-nearest neighbors (KNN)**:

```
# Load libraries
import numpy as np
from fancyimpute import KNN
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs
# Make a simulated feature matrix
features, _ = make_blobs(n_samples = 1000, n_features = 2, random_state = 1)
# Standardize the features
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)
# Replace the first feature's first value with a missing value
true_value = standardized_features[0,0]
standardized_features[0,0] = np.nan
# Predict the missing values in the feature matrix
features_knn_imputed = KNN(k=5, verbose=0).complete(standardized_features)
# Compare true and imputed values
print("True Value:", true_value)
print("Imputed Value:", features_knn_imputed[0,0])
```

OUTPUT:



```
True Value: 0.8730186114
Imputed Value: 1.09553327131
```

Imputation using the KNNimputer()

If you have a small amount of data, **predict the missing values using KNNimputer() of scikit-learn class**:

```
# import necessary libraries
import numpy as np
import pandas as pd

# import the KNNimputer class
from sklearn.impute import KNNImputer

# create dataset for marks of a student
dict = {'Maths':[80, 90, np.nan, 95],
        'Chemistry': [60, 65, 56, np.nan],
        'Physics':[np.nan, 57, 80, 78],
        'Biology' : [78,83,67,np.nan]}
# creating a data frame from the list
Before_imputation =
pd.DataFrame(dict)
#print dataset before imputaion
print("Data Before performing
imputation\n", Before_imputation)
# create an object for KNNImputer
imputer = KNNImputer(n_neighbors=2)
After_imputation =
imputer.fit_transform(Before_imputatio
n)
# print dataset after performing the
operation
print("\n\nAfter performing
imputation\n",After_imputation)
```

Imputation using the KNNimputer()

If you have a small amount of data, **predict the missing values using KNNimputer() of scikit-learn class**:

OUTPUT:

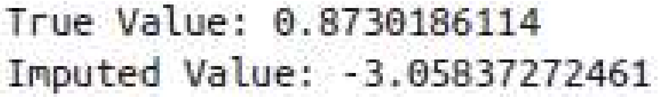
```
Data Before performing imputation
   Maths  Chemistry  Physics  Biology
0   80.0      60.0      NaN     78.0
1   90.0      65.0     57.0     83.0
2    NaN      56.0     80.0     67.0
3   95.0       NaN     78.0      NaN


After performing imputation
 [[80.  60.  68.5 78. ]
 [90.  65.  57.  83. ]
 [87.5 56.  80.  67. ]
 [95.  58.  78.  72.5]]
```

Imputing Missing Values

Alternatively, we can use **scikit-learn's Imputer module** to fill in missing values with the feature's mean, median, or most frequent value. However, we will typically get worse results than KNN:

```
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs
from sklearn.preprocessing import Imputer
# make fake data
features, _ = make_blobs(n_samples = 1000, n_features = 2, random_state = 1)
# standardize the features
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)
# replace the first feature's first value with a missing value
true_value = standardized_features[0, 0]
standardized_features[0,0] = np.nan
# create imputer
mean_imputer = Imputer(strategy="mean", axis=0)
# impute values
features_mean_imputed = mean_imputer.fit_transform(features)
# compare true and imputed values
print("True Value: {}".format(true_value))
print("Imputed Value: {}".format(features_mean_imputed[0,0]))
```

OUTPUT:



```
True Value: 0.8730186114
Imputed Value: -3.05837272461
```

Imputing Missing Values

Alternatively, we can use **scikit-learn's Imputer module** to fill in missing values with the feature's mean, median, most frequent (mode), or, a constant value:

```
import pandas as pd
import numpy as np
from sklearn.impute import
SimpleImputer
students = [[85, 'M', 'verygood'],
        [95, 'F', 'excellent'],
        [75, None,'good'],
        [np.NaN, 'M', 'average'],
        [70, 'M', 'good'],
        [np.NaN, None, 'verygood'],
        [92, 'F', 'verygood'],
        [98, 'M', 'excellent']]
dfstd = pd.DataFrame(students)
dfstd.columns = ['marks', 'gender', 'result']
print("Data Before performing
imputation\n",dfstd)
# Missing values is represented using
NaN and hence specified. If it
# is empty field, missing values will be
specified as follows:

#
# Imputing with mean value
imputer =
SimpleImputer(missing_values=np.Na
N, strategy='mean')

dfstd.marks =
imputer.fit_transform(dfstd['marks'].va
lues.reshape(-1,1))[:,0]

print("\n\nAfter performing
imputation\n",dfstd)
```

Imputing Missing Values…contd.

OUTPUT:

```
Data Before performing imputation
   marks gender     result
0   85.0      M   verygood
1   95.0      F  excellent
2   75.0   None       good
3    NaN      M    average
4   70.0      M       good
5    NaN   None   verygood
6   92.0      F   verygood
7   98.0      M  excellent
```

```
After performing imputation
       marks gender     result
0  85.000000      M   verygood
1  95.000000      F  excellent
2  75.000000   None       good
3  85.833333      M    average
4  70.000000      M       good
5  85.833333   None   verygood
6  92.000000      F   verygood
7  98.000000      M  excellent
```

Imputing Missing Values…contd.

Other Options:

```
# Imputing with median value
#
imputer = SimpleImputer(missing_values=np.NaN, strategy='median')
#
# Imputing with most frequent / mode value
#
imputer = SimpleImputer(missing_values=np.NaN, strategy='most_frequent')
#
# Imputing with constant value; The command below replaces the missing
# value with constant value such as 80
#
imputer = SimpleImputer(missing_values=np.NaN, strategy='constant',
fill_value=80)
```

SIMPLE LINEAR REGRESSION

Simple linear regression is a statistical technique used for finding the existence of an association relationship between a dependent variable (aka response variable or outcome variable) and an independent variable (aka explanatory variable, predictor variable or feature).

We can only establish that change in the value of the outcome variable (*Y) is associated with change in the value of feature X, that is, regression* technique cannot be used for establishing causal relationship between two variables.

Regression is one of the most popular supervised learning algorithms in predictive analytics. A regression model requires the knowledge of both the outcome and the feature variables in the training dataset.

The following are a **few examples of simple and multiple linear regression** problems:

1. A hospital may be interested in finding how the total cost of a patient for a treatment varies with the body weight of the patient.

2. Insurance companies would like to understand the association between healthcare costs and ageing.

SIMPLE LINEAR REGRESSION…

3. An organization may be interested in finding the relationship between revenue generated from a product and features such as the price, money spent on promotion, competitors' price, and promotion expenses.

4. Restaurants would like to know the relationship between the customer waiting time after placing the order and the revenue.

5. E-commerce companies such as Amazon, BigBasket, and Flipkart would like to understand the relationship between revenue and features such as
(a) Number of customer visits to their portal.
(b) Number of clicks on products.
(c) Number of items on sale.
(d) Average discount percentage.

6. Banks and other financial institutions would like to understand the impact of variables such as unemployment rate, marital status, balance in the bank account, rain fall, etc. on the percentage of non-performing assets (NPA).

STEPS IN BUILDING A REGRESSION MODEL

STEP 1: Collect/Extract Data:

The first step in building a regression model is to collect or extract data on the dependent (outcome) variable and independent (feature) variables from different data sources.

STEP 2: Pre-Process the Data:

It is essential to ensure the quality of the data for issues such as reliability, completeness, usefulness, accuracy, missing data, and outliers.

STEP 3: Dividing Data into Training and Validation Datasets:

The data is divided into two subsets (sometimes more than two subsets): training dataset and validation or test dataset. The proportion of training dataset is usually between 70% and 80% of the data and the remaining data is treated as the validation data. The subsets may be created using random/stratified sampling procedure.

STEP 4: Perform Descriptive Analytics or Data Exploration:

It is always a good practice to perform descriptive analytics before moving to building a predictive analytics model. Descriptive statistics will help us to understand the variability in the model and visualization of the data through graphs.

STEPS IN BUILDING A REGRESSION MODEL

STEP 5: Build the Model:
The model is built using the training dataset to estimate the regression parameters. The method of **Ordinary Least Squares (OLS)** is used to estimate the regression parameters.

STEP 6: Perform Model Diagnostics:
Regression is often misused since many times the modeler fails to perform necessary diagnostics tests before applying the model. Before it can be applied, it is necessary that the model created is validated for all model assumptions including the definition of the function form. If the model assumptions are violated, then the modeler must use remedial measure.

STEP 7: Validate the Model and Measure Model Accuracy:
A major concern in analytics is over-fitting, that is, the model may perform very well on the training dataset, but may perform badly in validation dataset. It is important to ensure that the model performance is consistent on the validation dataset as is in the training dataset. In fact, the model may be cross validated using multiple training and test datasets.

STEP 8: Decide on Model Deployment
The final step in the regression model is to develop a deployment strategy in the form of actionable items and business rules that can be used by the organization.

BUILDING SIMPLE LINEAR REGRESSION MODEL

Simple Linear Regression (SLR) is a statistical model in which there is only one independent variable (or feature) and the functional relationship between the outcome variable and the regression coefficient is linear. Linear regression implies that the mathematical function is linear with respect to regression parameters.

One of the functional forms of SLR is as follows:

$$Y = b_0 + b_1 X + e$$

For a dataset with *n observations (Xi, Yi), where i = 1, 2, ..., n, the above functional form can be written* as follows:

$$Y_i = b_0 + b_1 X_i + e_i$$

where Y_i *is the value of* i^{th} *observation of the dependent variable (outcome variable) in the sample,* X_i *is* the value of i^{th} *observation of the independent variable or feature in the sample,* e_i *is the random error* (also known as residuals) in predicting the value of Y_i, b_0 *and* b_1 *are the regression parameters (or regression* coefficients or feature weights).

The regression relationship stated above is a statistical relationship, and so is not exact, unlike a mathematical relationship, and thus the error terms e_i. *The above Equation can be written as*

$$e_i = Y_i - b_0 - b_1 X_i$$

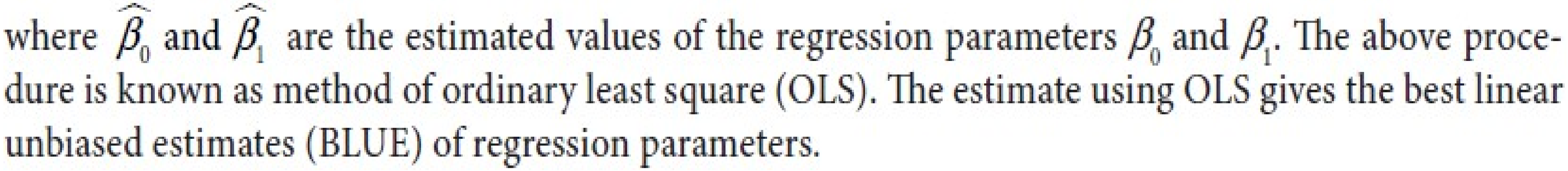
The regression parameters b_0 *and* b_1 *are estimated by minimizing the sum of squared errors (SSE).*

$$\text{SSE} = \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{n} (Y_i - \beta_0 - \beta_1 X_i)$$

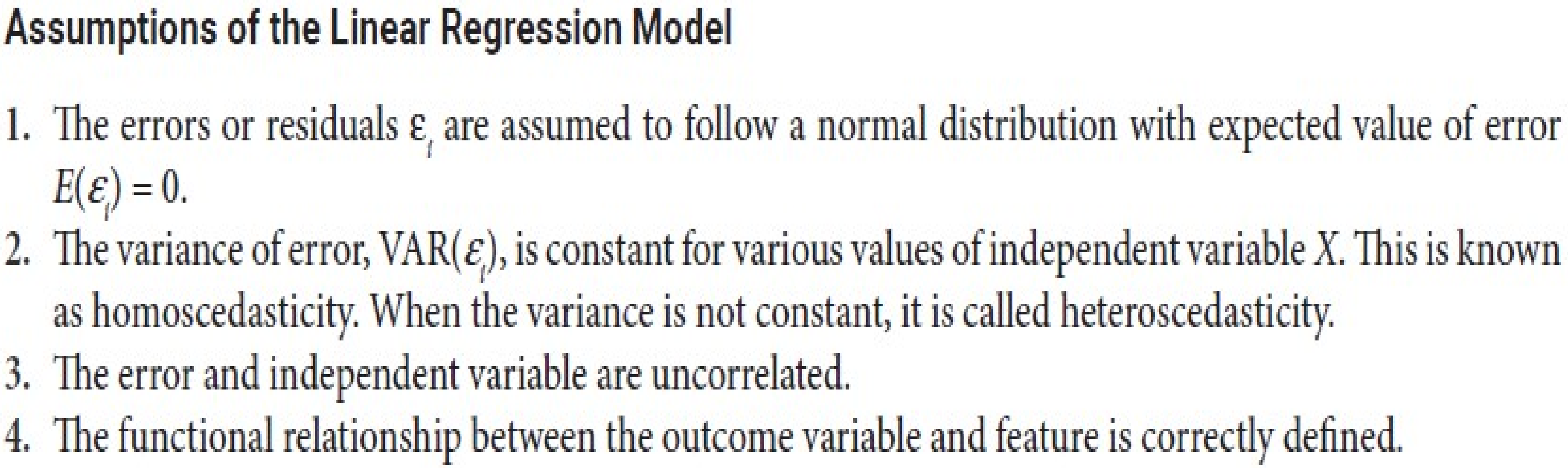
The estimated values of regression parameters are given by taking partial derivative of SSE with respect to b_0 *and* b_1 *and solving the resulting equations for the regression parameters. The estimated parameter* values are given by:

$$\widehat{\beta_1} = \sum_{i=1}^{n} \frac{(X_i - \overline{X})\,(Y_i - \overline{Y})}{(X_i - \overline{X})^2}$$

$$\widehat{\beta_0} = \overline{Y} - \beta_1 \overline{X}$$

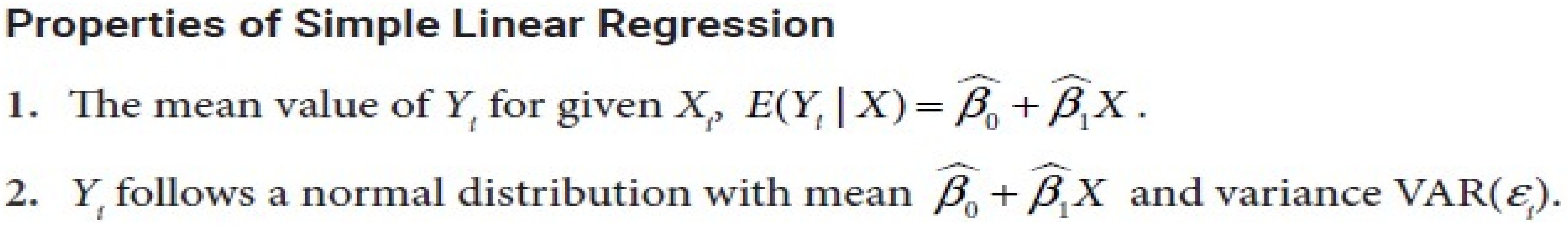


where $\widehat{\beta_0}$ and $\widehat{\beta_1}$ are the estimated values of the regression parameters β_0 and β_1. The above procedure is known as method of ordinary least square (OLS). The estimate using OLS gives the best linear unbiased estimates (BLUE) of regression parameters.



Assumptions of the Linear Regression Model

1. The errors or residuals ε_i are assumed to follow a normal distribution with expected value of error $E(\varepsilon_i) = 0$.
2. The variance of error, VAR(ε_i), is constant for various values of independent variable X. This is known as homoscedasticity. When the variance is not constant, it is called heteroscedasticity.
3. The error and independent variable are uncorrelated.
4. The functional relationship between the outcome variable and feature is correctly defined.



Properties of Simple Linear Regression

1. The mean value of Y_i for given X_i, $E(Y_i \mid X) = \widehat{\beta}_0 + \widehat{\beta}_1 X$.
2. Y_i follows a normal distribution with mean $\widehat{\beta}_0 + \widehat{\beta}_1 X$ and variance VAR(ε_i).

Correlation and Causation

Although correlation helps us determine the degree of relationship between two or more variables, it does not tell us about the cause and effect relationship. A high degree of correlation does not always necessarily mean a relationship of cause and effect exists between variables. Note that correlation does not imply causation, although the existence of causation always implies correlation.

Let’s understand this better with examples:

- More firemen’s presence during a fire instance signifies that the fire is big, but the fire is not caused by firemen.

- When one sleeps with shoes on, one is likely to get a headache. This may be due to alcohol intoxication.

Correlation and Causation…contd.

The significant degree of correlation in the preceding examples may be due to the following reasons:

- Small samples are prone to show a higher correlation due to pure chance.

- Variables may be influencing each other, so it becomes hard to designate one as the cause and the other the effect.

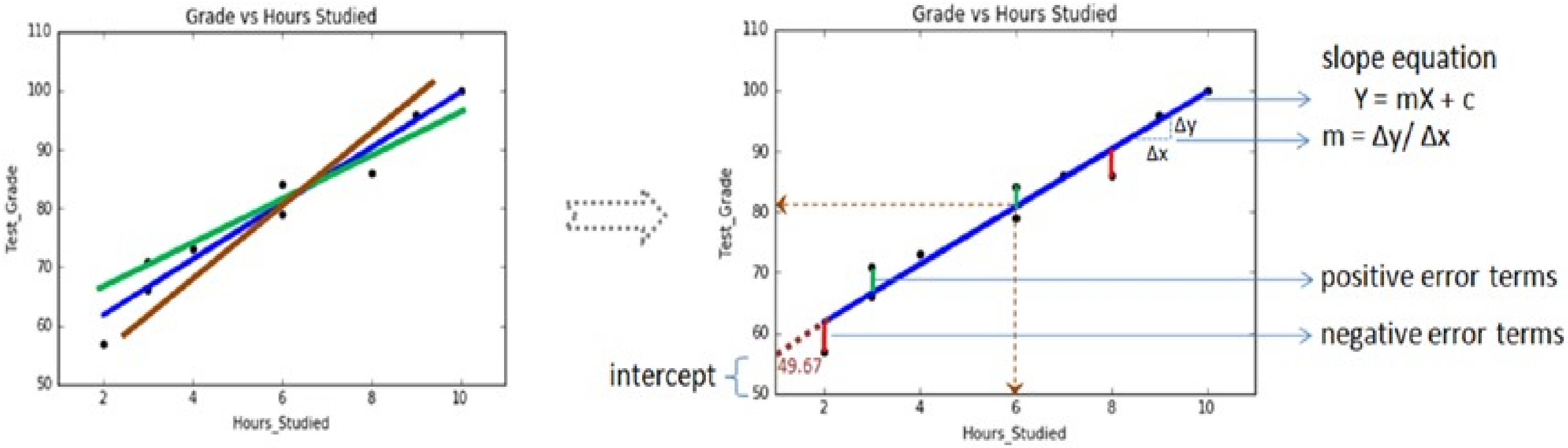
- Correlated variables may be influenced by one or more other related variables.

The domain knowledge or involvement of a subject matter expert is very important, to ascertain the correlation due to causation.

Fitting a Slope

Let’s try to fit a slope line through all the points such that the **error or residual** (i.e., the distance of the line from each point) is best possible minimal.

The error could be positive or negative based on its location from the slope, because of which if we take a simple sum of all the errors it will be zero. So we should square the error to get rid of negativity and then sum the squared error. Hence, the slope is also referred to as the least squares line.



slope equation

Y = mX + c

m = Δy/ Δx

positive error terms

negative error terms

intercept

Different possible slopes

Best slope that gives minimal sum of squared error

Linear Regression Model Components

Fitting a Slope…contd.

- The slope equation is given by Y = mX + c, where Y is the predicted value for a given x value.

- m is the change in y, divided by change in x (i.e., m is the slope of the line for the x variable and it indicates the steepness at which it increases with every unit increase in x variable value).

- c is the intercept, which indicates the location or point on the axis where it intersects; in this case it is 49.67. The intercept is a constant that represents the variability in Y that is not explained by the X. It is the value of Y when X is zero.

Together the slope and intercept define the linear relationship between the two variables and can be used to predict or estimate an average rate of change.

Fitting a Slope…contd.

Using this relation for a new student, we can determine the score based on his/her study hours. Say a student is planning to study an overall 6 hours in preparation for the test.

Simply drawing a connecting line from the x-axis and y-axis to the slope shows that there is a possibility of the student scoring 80.

We can use the slope equation to predict the score for any given hours of study.

In this case, the test grade is the dependent variable, denoted by “Y” and hours studied is the independent variable or predictor, denoted by “X.”

Let’s use the **linear regression function from the Scikit-learn library** to find the values of m (x’s coefficient) and c (intercept).

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
# importing linear regression function
import sklearn.linear_model as lm
# function to calculate r-squared, MAE, RMSE
from sklearn.metrics import r2_score , mean_absolute_error, mean_squared_error
%matplotlib inline

# Load data
df = pd.read_csv('Data/Grade_Set_1.csv')
print(df)
# Simple scatter plot
df.plot(kind='scatter', x='Hours_Studied', y='Test_Grade', title='Grade vs Hours Studied')
plt.show()
# check the correlation between variables
print("Correlation Matrix: ")
print(df.corr())
# Lets plot the distribution
fit = stats.norm.pdf(df.Test_Grade, np.mean(df.Test_Grade), np.std(df.Test_Grade))
plt.plot(df.Test_Grade,fit,'-o')
plt.hist(df.Test_Grade)
plt.show()
```

```
# Create linear regression object
lr = lm.LinearRegression()
x= df.Hours_Studied[:, np.newaxis] # independent variable
y= df.Test_Grade.values            # dependent variable
# Train the model using the training sets
lr.fit(x, y)
print("Intercept: ", lr.intercept_)
print("Coefficient: ", lr.coef_)
# manual prediction for a given value of x
print("Manual prediction :", 49.67777777777776 + 5.01666667*6)
# predict using the built-in function
print("Using predict function: ", lr.predict([[6]]))
# plotting fitted line
plt.scatter(x, y,  color='black')
plt.plot(x, lr.predict(x), color='blue', linewidth=3)
plt.title('Grade vs Hours Studied')
plt.ylabel('Test_Grade')
plt.xlabel('Hours_Studied')
```

OUTPUT:

```
Intercept: 49.67777777777776
Coefficient: [5.01666667]
Manual prediction : 79.77777779777776
Using predict function: [79.77777778]
```

INTERPRETATION:

Let’s put the appropriate values in the slope equation ($m * X + c = Y$),

$$5.01 * 6 + 49.67 = 79.77;$$

that means a student studying 6 hours has the probability of scoring a 79.77 test grade.

Note that if X is zero, the value of Y will be 49.67. That means even if the student does not study, there is a possibility that the score will be 49.67.

This signifies that there are other variables that have a causation effect on the score that we do not have access to currently.

How Good Is Your Model?

There are three metrics widely used for evaluating linear model performance:

- **R-Squared**
- **RMSE**
- **MAE**

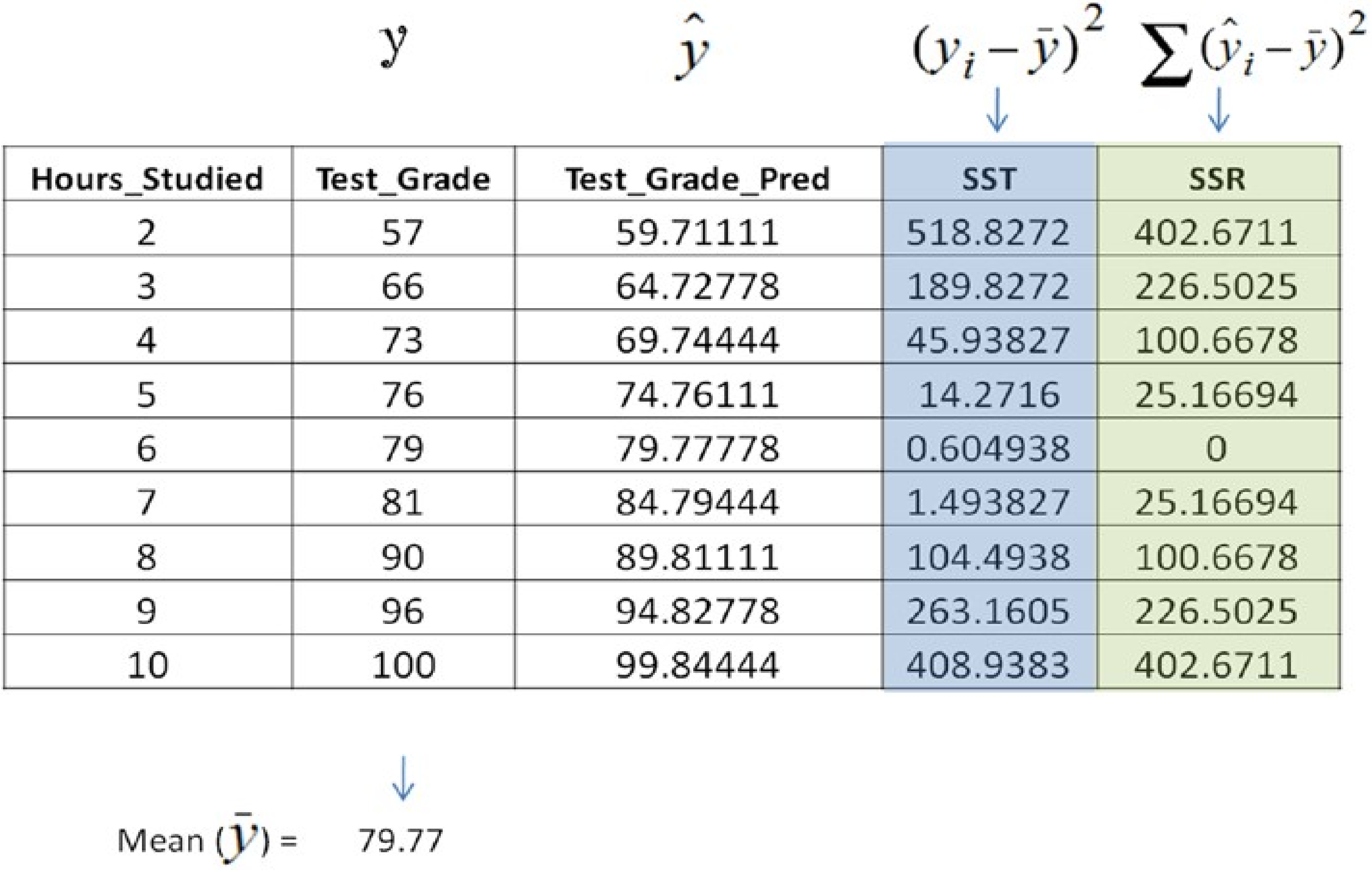
R-Squared for Goodness of fit

The R-Squared metric is the most popular practice of evaluating how well your model fits the data.

R-Squared value designates the total proportion of variance in the dependent variable explained by the independent variable.

It is a value between 0 and 1; the value toward 1 indicates a better model fit.

Sample Table for R-Squared Calculation

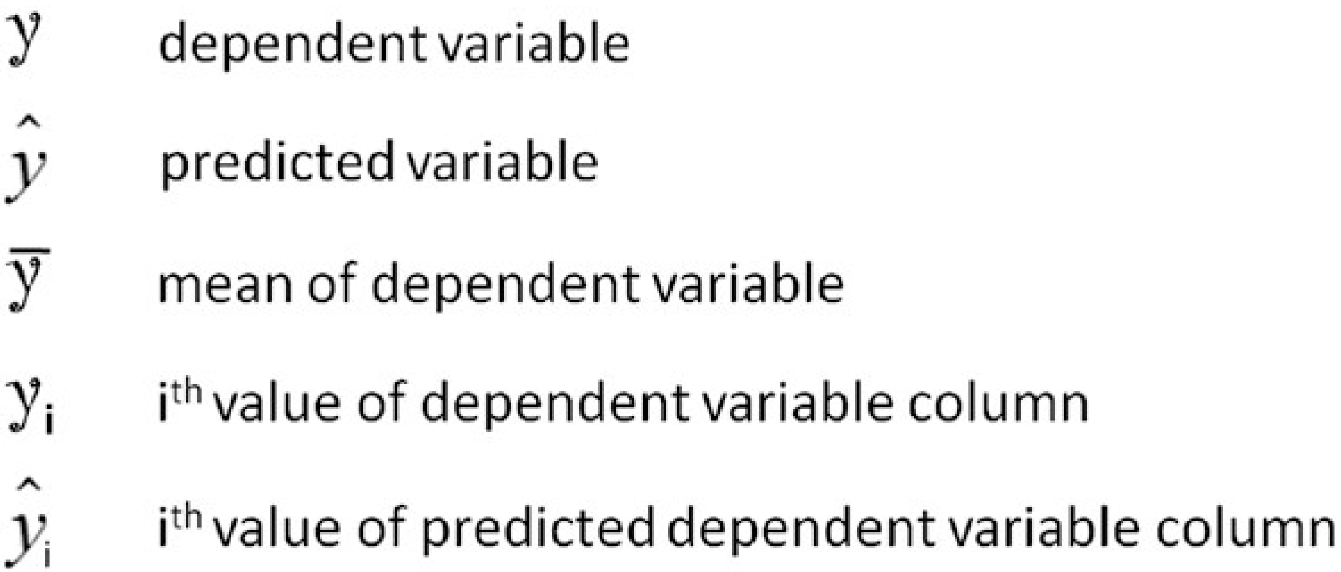


	y	$\hat{y}$	$(y_i - \bar{y})^2$ ↓	$\sum(\hat{y}_i - \bar{y})^2$ ↓
Hours_Studied	Test_Grade	Test_Grade_Pred	SST	SSR
2	57	59.71111	518.8272	402.6711
3	66	64.72778	189.8272	226.5025
4	73	69.74444	45.93827	100.6678
5	76	74.76111	14.2716	25.16694
6	79	79.77778	0.604938	0
7	81	84.79444	1.493827	25.16694
8	90	89.81111	104.4938	100.6678
9	96	94.82778	263.1605	226.5025
10	100	99.84444	408.9383	402.6711

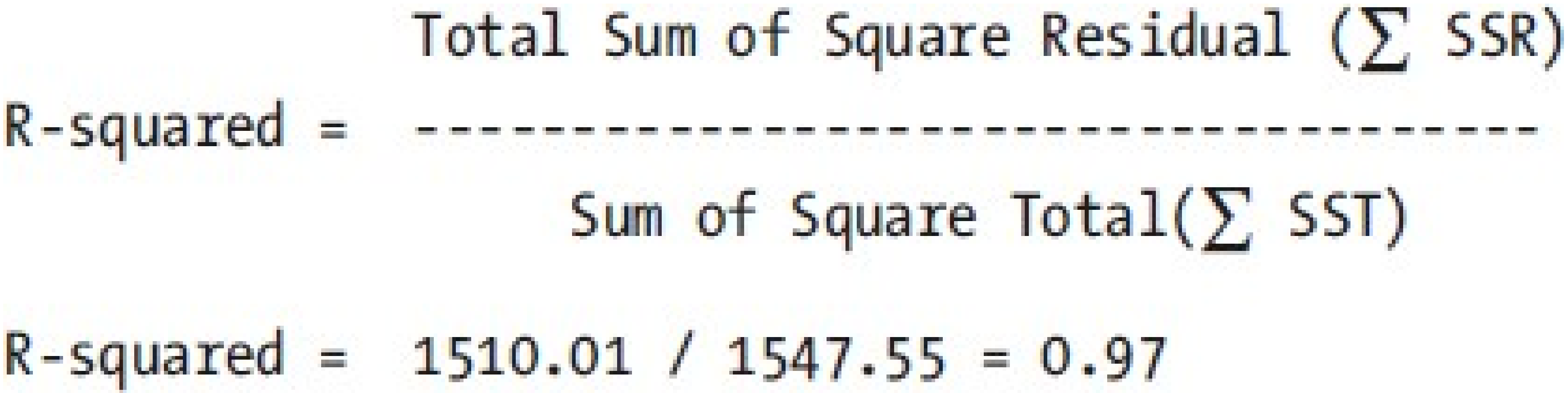
↓

Mean ($\bar{y}$) = 79.77

Where,



y	dependent variable
$\hat{y}$	predicted variable
$\overline{y}$	mean of dependent variable
y_i	ith value of dependent variable column
$\hat{y}_i$	ith value of predicted dependent variable column



```
              Total Sum of Square Residual (∑ SSR)
R-squared =  ------------------------------------
                  Sum of Square Total(∑ SST)

R-squared =  1510.01 / 1547.55 = 0.97
```

In this case, **R-Squared can be interpreted as 97% of the variability in the dependent variable (test score) and can be explained by the independent variable (hours studied)**.

Root Mean Squared Error (RMSE)

This is the square root of the mean of the squared errors. **RMSE indicates how close the predicted values are to the actual values**; hence, lower RMSE value signifies that the model performance is good. One of the key properties of RMSE is that the unit will be the same as the target variable.

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i-\hat{y}_i\right)^2}$$

Mean Absolute Error (MAE)

This (MAE) is the mean or average of the absolute value of the errors, that is, the predicted - actual.

$$\frac{1}{n}\sum_{i=1}^{n}\left|y_i-\hat{y}_i\right|$$

```
# Let's check the performance of fitted model through R-squared

# add predict value to the data frame
df['Test_Grade_Pred'] = lr.predict(x)

# Manually calculating R Squared
df['SST'] = np.square(df['Test_Grade'] - df['Test_Grade'].mean())
df['SSR'] = np.square(df['Test_Grade_Pred'] - df['Test_Grade'].mean())

print("Sum of SSR:", df['SSR'].sum())
print("Sum of SST:", df['SST'].sum())

print(df)
df.to_csv('r-squared.csv', index=False)

print("R Squared using manual calculation: ", df['SSR'].sum() / df['SST'].sum())

# Using built-in function
print("R Squared using built-in function: ", r2_score(df.Test_Grade,  df.Test_Grade_Pred))
print("Mean Absolute Error: ", mean_absolute_error(df.Test_Grade, df.Test_Grade_Pred))
print("Root Mean Squared Error: ", np.sqrt(mean_squared_error(df.Test_Grade,
        df.Test_Grade_Pred)))
```

Outliers

Lets introduce a outlier i.e., a student has studied 5 hours and scored 100. Assume that this student is has higher IQ than others in this group.
Notice the drop in R-squared value. So it is important to apply business logic to avoid including outliers in the training data set to generalize the model and increase accuracy.

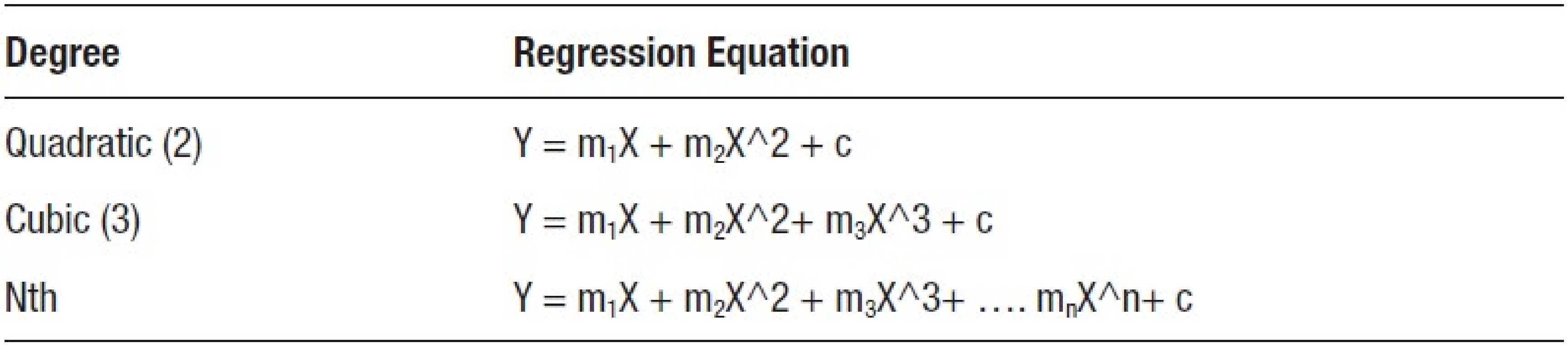
```
# Create linear regression object
lr = lm.LinearRegression()
# Load data
df =
pd.read_csv('F:/Practical__PYTHON/basic_materials/2__code__data/Chapter_3_C
ode/Code/Data/Grade_Set_1.csv')
df.loc[9] = np.array([5, 100])
x= df.Hours_Studied[:, np.newaxis] # independent variable
y= df.Test_Grade.values            # dependent variable
# Train the model using the training sets
lr.fit(x, y)
print("Intercept: ", lr.intercept_)
print("Coefficient: ", lr.coef_)
```

```
# manual prediction for a given value of x
print("Manual prediction :", 54.4022988505747 + 4.64367816*6)
# predict using the built-in function
print("Using predict function: ", lr.predict([[6]]))
# plotting fitted line
plt.scatter(x, y,  color='black')
plt.plot(x, lr.predict(x), color='blue', linewidth=3)
plt.title('Grade vs Hours Studied')
plt.ylabel('Test_Grade')
plt.xlabel('Hours_Studied')
# add predict value to the data frame
df['Test_Grade_Pred'] = lr.predict(x)
# Using built-in function
print("R Squared : ", r2_score(df.Test_Grade,  df.Test_Grade_Pred))
print("Mean Absolute Error: ", mean_absolute_error(df.Test_Grade,
    df.Test_Grade_Pred))
print("Root Mean Squared Error: ", np.sqrt(mean_squared_error(df.Test_Grade,
    df.Test_Grade_Pred)))
```

Polynomial Regression

It is a form of higher order linear regression modeled between dependent and independent variables as an nth degree polynomial. Although it’s linear, it can fit curves better. Essentially we’ll be introducing higher order degree variables of the same independent variable in the equation.

Polynomial Regression of Higher Degrees



Degree	Regression Equation
Quadratic (2)	$Y = m_1X + m_2X\text{\textasciicircum}2 + c$
Cubic (3)	$Y = m_1X + m_2X\text{\textasciicircum}2+ m_3X\text{\textasciicircum}3 + c$
Nth	$Y = m_1X + m_2X\text{\textasciicircum}2 + m_3X\text{\textasciicircum}3+ \ldots.\ m_nX\text{\textasciicircum}n+ c$

```
x = np.linspace(-3,3,1000)

# Plot subplots
fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(nrows=2, ncols=3)

ax1.plot(x, x)
ax1.set_title('linear')
ax2.plot(x, x**2)
ax2.set_title('degree 2')
ax3.plot(x, x**3)
ax3.set_title('degree 3')
ax4.plot(x, x**4)
ax4.set_title('degree 4')
ax5.plot(x, x**5)
ax5.set_title('degree 5')
ax6.plot(x, x**6)
ax6.set_title('degree 6')

# tidy layout
plt.tight_layout()
```

Let’s consider another set of students average test grade scores and their respective average studied hours for similar IQ students.

```
# Load data
df =
pd.read_csv('F:/Practical__PYTHON/basic_materials/2__code__data/Chapter_3_C
ode/Code/Data/Grade_Set_2.csv')

print(df)

# Simple scatter plot
df.plot(kind='scatter', x='Hours_Studied', y='Test_Grade', title='Grade vs Hours
Studied')

# check the correlation between variables
print("Correlation Matrix: ")
df.corr()
```

```
# Create linear regression object
lr = lm.LinearRegression()

x= df.Hours_Studied[:, np.newaxis]          # independent variable
y= df.Test_Grade                            # dependent variable

# Train the model using the training sets
lr.fit(x, y)
print("Intercept: ", lr.intercept_)
print("Coefficient: ", lr.coef_)

# manual prediction for a given value of x
print("Manual prdiction :", 7.27106067219556 + 7.25447403*6)

# predict using the built-in function
print("Using predict function: ", lr.predict([[6]]))

# plotting fitted line
plt.scatter(x, y,  color='black')
plt.plot(x, lr.predict(x), color='blue', linewidth=3)
plt.title('Grade vs Hours Studied')
plt.ylabel('Test_Grade')
plt.xlabel('Hours_Studied')

print("R Squared: ", r2_score(y, lr.predict(x)))
```

NumPy's vander function will return powers of the input vector:

```
lr = lm.LinearRegression()

x= df.Hours_Studied        # independent variable
y= df.Test_Grade           # dependent variable

# NumPy's vander function will return powers of the input vector
for deg in [1, 2, 3, 4, 5]:
   lr.fit(np.vander(x, deg + 1), y);
   y_lr = lr.predict(np.vander(x, deg + 1))
   plt.plot(x, y_lr, label='degree ' + str(deg));
   plt.legend(loc=2);
   print("R-squared for degree " + str(deg) + " = ",  r2_score(y, y_lr))
plt.plot(x, y, 'ok')
```

sklearn provides a function to generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree :

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

x= df.Hours_Studied[:, np.newaxis] # independent variable
y= df.Test_Grade                   # dependent variable

degree = 3
model = make_pipeline(PolynomialFeatures(degree), lr)


model.fit(x, y)

plt.scatter(x, y,  color='black')
plt.plot(x, model.predict(x), color='green')
plt.title('Grade vs Hours Studied')
plt.ylabel('Test_Grade')
plt.xlabel('Hours_Studied')

print("R Squared using built-in function: ", r2_score(y, model.predict(x)))
```

Logistic Regression

Let’s consider a use case where we have to predict students test outcome: pass (1) or fail (0) based on hours studied. In this case, the outcome to be predicted is discrete. Let’s build a linear regression and try to use a threshold: anything over some value is a pass, else fail.

Feature Selection For Machine Learning

1. Univariate Selection.

2. Recursive Feature Elimination.

3. Principle Component Analysis.

4. Feature Importance.

Feature Selection

Feature selection is a process where you automatically select those features in your data that contribute most to the prediction variable or output in which you are interested.

Having irrelevant features in your data can decrease the accuracy of many models, especially linear algorithms like linear and logistic regression. Three benefits of performing feature selection before modeling your data are:

- **Reduces Overfitting:** Less redundant data means less opportunity to make decisions based on noise.

- **Improves Accuracy:** Less misleading data means modeling accuracy improves.

- **Reduces Training Time:** Less data means that algorithms train faster.

1. Univariate Selection

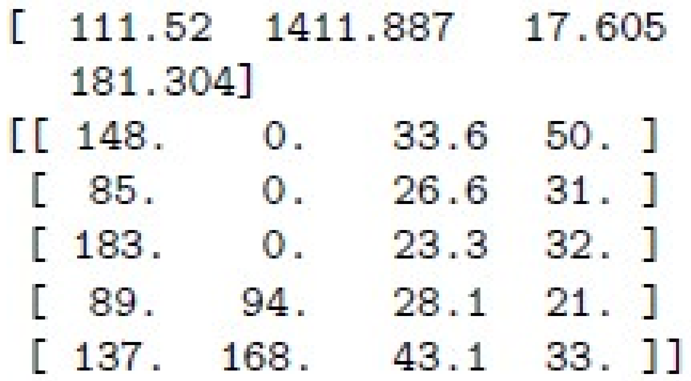
Statistical tests can be used to select those features that have the strongest relationship with the output variable.

The **scikit-learn library** provides the **SelectKBest class** that can be used with a suite of different statistical tests to select a specific number of features. The following example uses the chi-squared (chi2) statistical test for non-negative features to select 4 of the best features from the **Pima Indians onset of diabetes** dataset.

```
# Feature Extraction with Univariate Statistical Tests (Chi-squared for classification)
from pandas import read_csv
from numpy import set_printoptions
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
# load data
filename = 'F:/Practical__PYTHON/datasets/pima-indians-diabetes.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
#print(dataframe)
X = array[:,0:8]
Y = array[:,8]
# feature extraction
test = SelectKBest(score_func=chi2, k=4)
fit = test.fit(X,Y)
# summarize scores
set_printoptions(precision=3)
print(fit.scores_)
features = fit.transform(X)
# summarize selected features
print(features[0:5,:])
```

OUTPUT:

The scores for each attribute and the 4 attributes chosen (those with the highest scores): plas, test, mass and age. We got the names for the chosen attributes by manually mapping the index of the 4 highest scores to the index of the attribute names.



```
[  111.52  1411.887   17.605   53.108 2175.565  127.669    5.393
   181.304]
[[ 148.     0.    33.6   50. ]
 [  85.     0.    26.6   31. ]
 [ 183.     0.    23.3   32. ]
 [  89.    94.    28.1   21. ]
 [ 137.   168.    43.1   33. ]]
```

2. Recursive Feature Elimination

The Recursive Feature Elimination (or RFE) works by recursively removing attributes and building a model on those attributes that remain. It uses the model accuracy to identify which attributes (and combination of attributes) contribute the most to predicting the target attribute.

You can learn more about the **RFE** class in the scikit-learn documentation:

https:// sklearn.feature_selection.RFE — scikit-learn 1.2.1 documentation

The following example uses RFE with the logistic regression algorithm to select the top 3 features. The choice of algorithm does not matter too much as long as it is skillful and consistent.

```
from pandas import read_csv
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
# load data
url = "F:/Practical__PYTHON/datasets/pima-indians-diabetes.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
model = LogisticRegression(solver='lbfgs')
rfe = RFE(model, n_features_to_select=3, step=1)
fit = rfe.fit(X, Y)
print("Num Features: %d" % fit.n_features_)
print("Selected Features: %s" % fit.support_)
print("Feature Ranking: %s" % fit.ranking_)
```

OUTPUT:

You can see that RFE chose the top 3 features as preg, mass and pedi. These are marked True in the support_ array and marked with a choice 1 in the ranking array. Again, you can manually map the feature indexes to the indexes of attribute names.

```
Num Features: 3
Selected Features: [ True False False False False True True False]
Feature Ranking: [1 2 4 5 6 1 1 3]
```

3. Principal Component Analysis

Principal Component Analysis (or PCA) uses linear algebra to transform the dataset into a compressed form.

Generally this is called a data reduction technique.

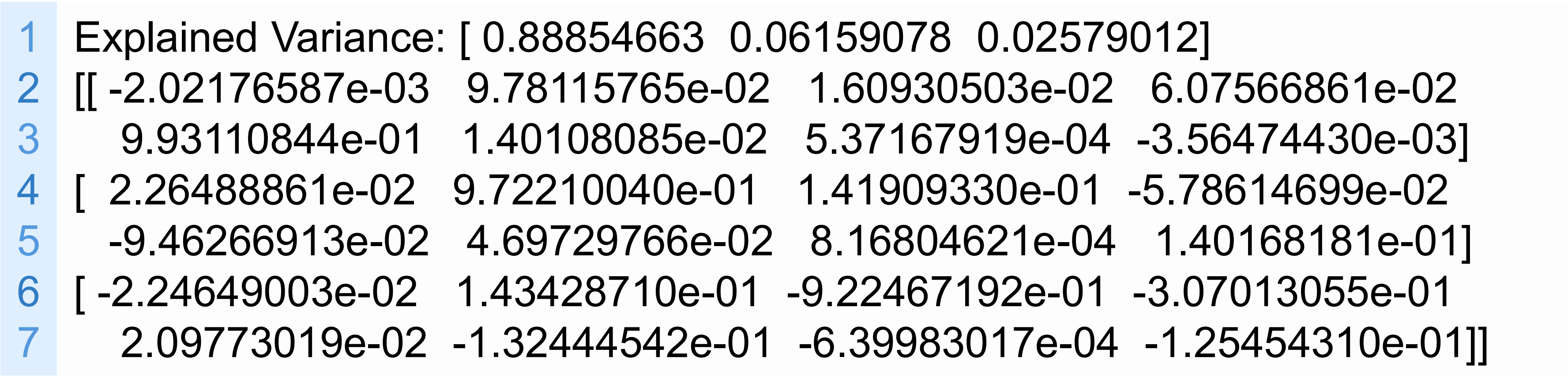
A property of PCA is that you can choose the number of dimensions or principal components in the transformed result.

In the following example, we use PCA and select 3 principal components. Learn more about the PCA class in scikit-learn by reviewing the API:

https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

```
# Feature Extraction with PCA
import numpy
from pandas import read_csv
from sklearn.decomposition import PCA
# load data
url = "F:\Practical__PYTHON\datasets/pima-indians-diabetes.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
pca = PCA(n_components=3)
fit = pca.fit(X)
# summarize components
print("Explained Variance: %s" % fit.explained_variance_ratio_)
print(fit.components_)
```

OUTPUT:



```
Explained Variance: [ 0.88854663  0.06159078  0.02579012]
[[ -2.02176587e-03   9.78115765e-02   1.60930503e-02   6.07566861e-02
    9.93110844e-01   1.40108085e-02   5.37167919e-04  -3.56474430e-03]
 [  2.26488861e-02   9.72210040e-01   1.41909330e-01  -5.78614699e-02
   -9.46266913e-02   4.69729766e-02   8.16804621e-04   1.40168181e-01]
 [ -2.24649003e-02   1.43428710e-01  -9.22467192e-01  -3.07013055e-01
    2.09773019e-02  -1.32444542e-01  -6.39983017e-04  -1.25454310e-01]]
```

You can see that the transformed dataset (3 principal components) bare little resemblance to the source data.

4. Feature Importance

Bagged decision trees like Random Forest and Extra Trees can be used to estimate the importance of features.

In the example below we construct a ExtraTreesClassifier classier for the Pima Indians onset of diabetes dataset.

You can learn more about the **ExtraTreesClassifier** class:

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html

```
# Feature Importance with Extra Trees Classifier
from pandas import read_csv
from sklearn.ensemble import ExtraTreesClassifier
# load data
url = "F:/Practical__PYTHON/datasets/pima-indians-diabetes.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
model = ExtraTreesClassifier(n_estimators=10)
model.fit(X, Y)
print(model.feature_importances_)
```

OUTPUT:

```
[0.11505474 0.24185013 0.09443805 0.0727363 0.08236756 0.1459499 0.12395058 0.12365273]
```

You can see that we are given an importance score for each attribute where the larger the score, the more important the attribute. The scores suggest at the importance of plas, age and mass.