Introduction to **Regression**

Regression

Regression is a supervised machine learning technique that helps in predicting continuous numerical values or quantity. For Example temperature, price and so on.

Regression Model can be a linear or a non-linear function. Let us understand the process of Linear Regression.

Suppose, a computer service centre records the data of time (in minutes) required in repairing a computer and number of faulty units. ([Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600395112628224197/web-hosted/assets/datasets1603366608907.zip) to download the dataset.)

Let us import the data.

*# Importing the required libraries*

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

%matplotlib inline

*# Reading the data from input csv file into a Pandas DataFrame*

computers = pd.read\_csv("datasets/computers.csv")

*# Printing the first 5 rows*

computers.head()

A white rectangular grid with black numbers

Description automatically generated

We want to predict 'Minutes' (target) from 'Units' (predictor) value.

To estimate the time taken by the service center to repair a computer, which of the following would be chosen?

Arithmetic mean = 97.21 minutes (Arithmetic mean is calculated as the sum of all values divided by the total number of values.)

Median = 96.50 minutes (Median is calculated as the middle most value in the range of values that we have, when arranged in an order.

Suppose you choose 'mean' of the 'Minutes' values as the function to estimate the time to repair a computer. This means, no matter how many faulty units you have, you will always predict 97.21 minutes as the time needed to repair the computer.

The following graph compares the various values of time taken to repair a computer (shown in the scatter plot) against the mean (represented by the horizontal line).

*# Calculating the mean.*

mean\_value = computers['Minutes'].mean()

*# Plotting the actual time taken for various values of number of faulty units.*

plt.scatter(computers['Units'],computers['Minutes'],color='blue', marker='o')

*# Plotting the mean*

plt.axhline(y=mean\_value,c="r")

*# Adding annotation, xlabel and ylabel.*

plt.annotate("Mean repair time",xy = (7.5,mean\_value+2))

plt.xlabel("Units")

plt.ylabel("Minutes")

A graph showing the difference between repair and maintenance

Description automatically generated

From the graph, it can be seen that, if 'mean' is used to predict the time taken to repair a computer, for some instances, a significant difference between the actual (observed) value and the predicted value of time is observed.

For example, the 2nd repair (index=2) took 29 minutes. If we use the mean to predict the time, we would have predicted the time as 97.21 minutes. Here the observed time is much lesser than the predicted time. On the other hand, consider the 13th repair. The observed time taken is 154 minutes which is larger than the predicted value by as much as 56.79 minutes.

This indicates that to predict the expected time taken to repair a computer, we may have to consider other factors as well.

**Understanding the association between variables:**

Let us now understand how the repair time varies based on the number of units to be replaced. For this we require a scatter plot, as shown below.

*# Plotting the actual time taken for various values of number of faulty units.*

plt.scatter(computers['Units'],computers['Minutes'],color='blue', marker='o')

# Plotting the mean

plt.axhline(y=mean\_value,c="r")

# Adding annotation, xlabel and ylabel.

plt.annotate("Mean repair time",xy = (7.5,mean\_value+2))

plt.xlabel("Units")

plt.ylabel("Minutes")

A graph with blue dots and red line

Description automatically generated

The above scatter plot suggests a linear association between the number of units replaced and the time taken to repair the computer. That means as the number of faulty units increases, the time taken to repair the computer also increases.

Regression Techniques:

To predict the value of time taken to repair a computer based on the number of units being replaced, a regression model can be built using Regression analysis.

Regression analysis is a statistical process for estimating the relationships between variables. It can be used to build a model to predict the value of the target variable from the predictor variables.

Mathematically, a regression model is represented as y= f(X), where y is the target or dependent variable and X is the set of predictors or independent variables (x1, x2, …, xn).

If a linear regression model involves only one predictor variable, it is called a **Simple Linear Regression model**.

f(X) = ß0 +ß1x1+ ∈

If a linear regression model involves multiple predictor variables, it is called a **Multiple Linear Regression model**.

f(X) = ß0 +ß1x1 +ß2x2+ ... +ßnxn+ ∈

The ß values are known as weights (ß0 is also called intercept and the subsequent ß1, ß2, etc. are called as coefficients). The error ,  ϵ is assumed to be normally distributed with a constant variance.

**Simple Linear Regression**

In the Regression example, we tried predicting the time to repair a computer from only a single variable i.e. the number of faulty units. Now, let us try using Simple Linear Regression Model.

Thus, the model can be framed as:

**Time taken to repair a computer = ß0+ (ß1\* Units being replaced) + ε**

We have to the find out the best values of (ß0, ß1) that can represent the true nature of the relationship between the number of faulty units in a computer and the time taken to repair the computer.

Creating Regression Models:

Let us speculate a few linear regression models that can predict the value of time taken to repair a computer.

**Model 0**: Time taken to repair a computer = 97.21 (i.e. mean)

**Model 1**: Time taken to repair a computer = 10 + (12 \* No. of Units being replaced)

**Model 2**: Time taken to repair a computer = 6 + (18 \* No. of Units being replaced)

minutes\_model0 = computers['Minutes'].mean()

minutes\_model1 = 10 + 12\*computers['Units']

minutes\_model2 = 6 + 18\*computers['Units']

* In Model 0, it is assumed that the time taken to repair a computer is constant. Hence, the expected time taken to repair a computer is predicted using the mean.
* In Model 1, ß0= 10,can be assumed as the startup time required to understand the repair work to be done on a given computer. The other coefficient, ß1 = 12 is assumed to represent the time required to replace a single unit of computer. Model 1 takes into account the initial time spent to understand the repair work along with the replacement time that every single component shall consume.
* Similarly in Model 2, ß0and ß1are chosen to be 6 and 18 respectively.

Visualizing the Speculated Regression Models:

Plots of Model 0 (red line), Model 1 (green line) and Model 2 (black line) along with the given data are as shown below.

fig,ax = plt.subplots()

*# Plotting the actual 'Minutes'*

ax.scatter(x="Units",y="Minutes",data=computers,label='actual repair time')

*# Plotting the Model0 predictions*

ax.plot(computers['Units'],computers['min\_model0'],color="red",label='model0')

*# Plotting the Model1 predictions*

ax.plot(computers['Units'],computers['min\_model1'],color="green",label='model1')

*# Plotting the Model2 predictions*

ax.plot(computers['Units'],computers['min\_model2'],color="black",label='model2')

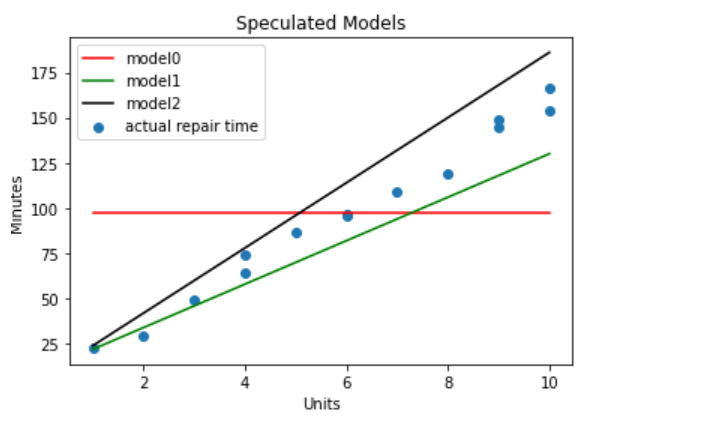
*# Adding xlabel, ylabel, title and legend*

ax.set\_ylabel("Minutes")

ax.set\_xlabel("Units")

ax.set\_title("Speculated Models")

ax.legend()



From the plot it can be observed that Model 1 and Model 2 predict the values of time taken to repair a computer better than Model 0.

**Analyzing the Speculated Models:**

The following code snippet shows the units replaced, the observed time taken, expected time taken (based on the model) and the difference between predicted and observed values for Model 0.

*# Validating Model0: Estimated time = mean('Minutes')*

*# Creating a Pandas DataFrame with 'Units', actual 'Minutes', predicted 'Minutes' by Model0, error in prediction by Model0.*

model0\_obs = pd.DataFrame({"Units":computers['Units'],

"Actual time":computers['Minutes'],

"Predicted time":computers['min\_model0'],

"Error":(computers['min\_model0'] - computers['Minutes'])})

model0\_obs *# Printing the DataFrame*

A table with numbers and a number on it

Description automatically generated

(Yue: part of the DF)

Here, it can be seen that the predicted values are significantly away from the observed values. This difference between predicted value and observed value is termed as an error in prediction.

**Analyzing Model 0**

The sum of individual errors for Model 0 is given below -

*# Sum of errors*

print(sum(model0\_obs['Error']))

1.4218….

In the previous observations from the graph and table, Model 0 (which used arithmetic mean) did not provide proper predictions, the total error came as 1.4e14, which is almost zero. This is because the positive and negative errors get cancelled with one another.

Therefore, to identify the total error of the model, we have to square the individual errors and sum them as shown below. This value is also known as Sum of Squared Errors or SSE.

*# Sum of squared errors*

sum(model0\_obs['Error']\*\*2)

**27768.3371…..**

**Analyzing Model 1**

The following code snippet shows the units replaced, the observed time taken, expected time taken (based on the model) and the difference between predicted and observed values for Model 1.

*# Creating a Pandas DataFrame with 'Units', actual 'Minutes', predicted 'Minutes' by Model1, error in prediction by Model1.*

*# Model1: Estimated time = 10 + 12\*(#Units) minutes*

model1\_obs = pd.DataFrame({"Units":computers.Units,

"Actual time":computers.Minutes,

"Predicted time":computers.min\_model1,el\_obs

"Error":(computers.min\_model1 - computers.Minutes)})

Model1\_obs

A table with numbers and text

Description automatically generated

(yue: partial display from the DF)

model1\_obs

Here, the predicted value is close to the observed value. This further strengthens the argument that Model 1 is a better predictor than Model 0. The sum of squared errors for Model 1 is also significantly lower than Model 0, as shown below.

*# SSE for Model1*

sum(model1\_obs.Error\*\*2)

4993

**Analyzing Model 2**

Similarly, the following code snippet shows the units replaced, the observed time taken, expected time taken (based on the model) and the difference between predicted and observed values for Model 2.

*# Creating a Pandas DataFrame with 'Units', actual 'Minutes', predicted 'Minutes' by Model2, error in prediction by Model2.*

*# Model2: Estimated time = 6 + 18\*(#Units) minutes*

model2\_obs = pd.DataFrame({"Units":computers.Units,

"Actual time":computers.Minutes,

"Predicted time":computers.min\_model2,

"Error":(computers.min\_model2 - computers.Minutes)})

model2\_obs

A table with numbers and text

Description automatically generated

(yue: partial display from the DF)

model2\_obs

The sum of squared errors for Model 2 is much lower than Model0, as shown below.

*# SSE for Model2*

sum(model2\_obs.Error\*\*2)

5001

**sAnalyzing the created models - Summary**

The table below shows the summary of the models learnt.

A screenshot of a computer

Description automatically generated

We can observe that the total sum of squared errors for Model 1 and Model 2 are close to each other and also significantly lower than the sum of squared errors of Model 0. With this in consideration, we can say that for the given sample dataset, Model 1 and Model 2 are better than Model 0 at predicting the expected value of time taken to repair a computer, based on the number of parts being replaced.

Here, we speculated 3 models and determined the expected values and the total sum of squared errors. The goal of linear regression is to create a model that predicts the value accurately and consequently has the lowest sum of squared errors (also known as least squares). Such a model is called as the best fit model.

m(model2\_*A screenshot of a math problem

Description automatically generated*

**Finding the Best Fit Model Manually**

By using the previously obtained formula, b0 and b1 for the given sample dataset is determined as approximately 4.162 and 15.509 respectively, as shown below.

x = computers.Units

y = computers.Minutes

xiyi = x\*y

n = len(computers)

xmean = computers.Units.mean()

ymean = computers.Minutes.mean()

numerator = xiyi.sum() - n\*xmean\*ymean

denominator = (x\*\*2).sum() - n\*(xmean\*\*2)

m = numerator/denominator

c = ymean - (m\*xmean)

print('intercept : ',c,' coefficient : ',m)

Therefore, the best fit model can be given as:

Estimated time taken to repair a computer = 4.16165 + 15.50877 \* Number of faulty units

*# Estimating time using the intercept and the coefficient of the best fit model*

min\_best\_fit\_model = c + m\*computers.Units

*# Adding the predicted values to the dataset*

computers['min\_best\_fit\_model'] = min\_best\_fit\_model

*# Printing the values predicted by the best fit model*

computers[["Units","Minutes","min\_best\_fit\_model"]]

A table of numbers and units

Description automatically generated

**Visualizing the Best Fit Model**

The plot of this model along with the given data is as shown below:

fig,ax = plt.subplots()

*# Plotting the actual target values*

ax.scatter(x="Units",y="Minutes",data=computers)

*# Plotting the target values predicted by the best fit model*

ax.plot(computers.Units,computers.min\_best\_fit\_model,color="red")

ax.set\_ylabel("Minutes")

ax.set\_xlabel("Units")

ax.set\_title("Best fit model line")

A diagram of a graph

Description automatically generated

This regression line (or function) gives the best estimate of the time taken to repair a computer, given the number of faulty units.

Sum of Squared Errors for the Best Fit Model:

The following table shows the number of units replaced, the observed time taken, expected time taken (based on the model) and the difference between predicted and observed values for the best fit model.

The sum of squared errors is approximately 348.8484.

*# Computing the individual errors for the best fit model*

best\_fit\_model\_obs = pd.DataFrame({"Units":computers.Units,

"Actual time":computers.Minutes,

"Predicted time":computers.min\_best\_fit\_model,

"Error":computers.min\_best\_fit\_model - computers.Minutes})

best\_fit\_model\_obs

A table of numbers with black numbers

Description automatically generated

*# Sum of Squared Errors for the best fit model*

sum(best\_fit\_model\_obs.Error\*\*2)

**Best Fit Model using sklearn.linear\_model.LinearRegression**

The best fit linear regression model can also be easily built in Python using the sklearn.linear\_model.LinearRegression class as shown below:

*# Setting the 'Units' column as the input data or predictor column*

X = computers[['Units']]

*# Setting the 'Minutes' column as the output data or target column*

y=computers['Minutes']

*# Importing the required class*

from sklearn.linear\_model import LinearRegression

*# Creating a linear regression model*

model = LinearRegression()

*# Fitting the model to the data i.e. building the model*

model.fit(X, y)

*# Fetching intercept and coefficient*

print("Intercept:", model.intercept\_)

print("Coefficients:",model.coef\_)

The intercept indicates the value of b0 and coefficients represents the remaining coefficient b1.

Thus the best fit model becomes:

**Estimated time taken to repair a computer = 4.161654 + (15.50877 x Units being replaced)**

A screenshot of a math problem

Description automatically generated

A white background with black text

Description automatically generated

Relation between SST, SSR and SSE

SST can also be expressed in terms of the sum of squared regression (SSR) and the sum of squared errors (SSE) as, SST = SSR + SSE.

A black and white text

Description automatically generated

In general, the residuals of a simple linear regression model can be visualized as shown below:

A diagram of a linear model

Description automatically generated

 The above figure depicts the simple linear regression line represented as ŷ = ß0 + ß1x. The figure also depicts the expected (predicted) value (xi,ŷi) and the observed (actual) value (xi,yi). The error is represented as êi = yi - ŷ.

Computing the Coefficient of Determination

The computation of R2 for the best fit model built on computers dataset is shown below.

SST = sum((computers.Minutes.mean() - computers.Minutes)\*\*2)

SST

*# 27768.357142857145*

SSE = sum(best\_fit\_model\_obs.Error\*\*2)

SSE

*# 348.848370927318*

*# 27419.508771929828*

Rsq = SSR/SST

Rsq

*# 0.9874371980620736*

Rsq1 = model.score(computers[["Units"]],y)

Rsq1

*# 0.9874371980620736*

As shown in the code, the **score()** of the model directly computes the R2value.

An R2 value of 0.9874372 indicates that 98.74 % of the variability observed in the dependent variable (time taken to repair computer) can be explained by variability in the predictor variable (units being replaced) in this sample data set.

R2can take values in the range 0 and 1. The higher the value of R2, the more useful is the model.

This concludes our discussion of Simple Linear Regression.

**Code:**[Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600400712212480221/web-hosted/assets/SimpleLinearRegression1603978171989.zip)[to download](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600400712212480221/web-hosted/assets/SimpleLinearRegression1603481088554.zip) the code discussed in Simple linear regression.

Exercise:

Use the Boston dataset (Click [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600401283375104232/web-hosted/assets/bostonhousing1603883195869.zip) to download the Boston dataset) and perform the following activities:

1.    Consider the column, ‘RM’, as predictor, and ‘MEDV’ as the target variable

2.    Visualize the association between the predictor and the target using scatter plot.

3.    Split the data into train and test datasets, in the ratio of 67:33.

4.    Build a Linear Regression model using training dataset, to predict the target variable.

5.    Observe the coefficient and intercept values for the model.

6.    Evaluate the model using mean squared error values and R-squared values on the training and the testing datasets.

**Multiple Linear Regression**

In Simple Linear Regression model, the dependent variable y is related to a single predictor variable x. This simple linear regression model having the dependent variable y can be described using:

A black and white text

Description automatically generated with medium confidence

However, there may be more than one predictor variables available. For example:

1. The volume of a tree trunk might depend on its height as well as girth.
2. The price of a house might depend on the number of bedrooms, the built-up area of the plot, the age of the house etc.
3. The height of a child might depend on age, weight, heights of the parents etc.

In order to predict the dependent variable based on multiple predictors, **Multiple Linear Regression model** is used. It is described as –



The regression coefficients *ßj*, where j = 1 to n.

## ****Multiple Linear Regression for the delivery dataset****

To understand Multiple Linear Regression, let us consider the delivery dataset . The delivery dataset provides information about the time taken to deliver a soft drink by vending machines.

Click [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600395710996480211/web-hosted/assets/delivery.csv) to download the delivery dataset.

Now, import the input data and take a look at the sample data.

1. *# read the data from input csv file*
2. delivery = pd.read\_csv("datasets/delivery.csv")
3. *# view sample data*
4. delivery.head()

A table with numbers and text

Description automatically generated

Let us take an insight into the data.

1. delivery.info()

A screenshot of a computer screen

Description automatically generated

The delivery dataset is a data frame with 25 observations on the following 3 variables.

* **n.prod**: Number of products stocked in a vending machine.
* **distance**: Distance traversed by the driver mechanism of the vending machine to deliver a soft drink.
* **delTime**: Time required to deliver the soft drink.

 To predict the time taken by a vending machine to deliver a soft drink, based on the number of products stocked and the distance traversed by the route driver of the vending machine. We shall try to fit a linear regression model as given below.



## ****Visualizing the delivery dataset****

A **scatter plot** can be used to get an insight on the nature of the relationship between the variables. Using a scatter plot, a linear association between the variables can be visually determined.

The scatter plot matrix of the variables - n.prod, distance and delTime is shown below.

Seaborn library, another visualization library is used in the below example.

1. *# Visualizing data*
2. import seaborn as sns
3. sns.pairplot(delivery)

A group of blue and white graphs

Description automatically generated

It is observed that the scatter plot matrix for the delivery time dataset shown above suggests a linear relationship among its variables – n.prod, distance and delTime.

**Building a Multiple Linear Regression model**

To build a multiple linear regression model, **sklearn.linear\_model.LinearRegression**to be used, as shown below:

1. *# importing the required module to build the model*
2. from sklearn.linear\_model import LinearRegression
3. *# Instantiating the algorithm for building the model*
4. model = LinearRegression()
5. *# selecting the predictors and targets*
6. X = delivery[["n.prod","distance"]]
7. y = delivery["delTime"]
8. *# building the model using fit() method*
9. model.fit(X,y)
10. print("Intercept:",model.intercept\_ ,"\nCoefficients:",model.coef\_)
11. *# Sample output for the above code*
12. *# Intercept: 2.3412311451922*
13. *# Coefficients: [1.61590721 0.01438483]*

And the best fit model to predict the delivery time based on n.prod and distance can be expressed as shown below.



## ****Visualizing Multiple Linear Regression model****

By now, you know that the Simple Linear Regression model represents a line and Multiple Regression Model represents a plane (in cases of two predictors) or a hyperplane (in cases of more than two predictors).

The best fit model obtained for predicting the delTime for the given sample dataset can be visualized as shown below.

1. *#Setting up the plot area*
2. fig = plt.figure(figsize=(7,7))
3. ax = fig.add\_subplot(111,projection='3d')#projection = '3d' is used to have 3D axes
4. *#3d scatterplot of the delivery dataset*
5. ax.scatter(xs = delivery["n.prod"], ys = delivery["distance"], zs=delivery["delTime"],
6. c='blue',
7. alpha=1,
8. marker='o')
9. ax.set\_xlabel("n.prod")
10. ax.set\_ylabel("distance")
11. ax.set\_zlabel("delTime")
12. *#Creating a mesh of x and y values to plot the regression plane*
13. x\_surf = np.arange(delivery["n.prod"].min(), delivery["n.prod"].max(), 1)
14. y\_surf = np.arange(delivery["distance"].min(), delivery["distance"].max(), 1)
15. x\_surf, y\_surf = np.meshgrid(x\_surf, y\_surf)
16. X\_mesh = pd.core.frame.DataFrame({'n.prod': x\_surf.ravel(), 'distance': y\_surf.ravel()})
17. *#Predicting the output of model for every point in the mesh*
18. out = model.predict(X\_mesh)
19. *#Plotting the regression plane*
20. ax.plot\_surface(x\_surf, y\_surf,
21. out.reshape(x\_surf.shape),
22. alpha = 0.4) *# apha => transparency of the surface*

The output is as shown below.

A graph with a blue and white line

Description automatically generated with medium confidence

**MultiCollinearity**

In a multiple regression model where two or more predictor variables are involved, it is possible that one predictor can be linearly predicted from the others, with a substantial degree of accuracy. In such a situation, the predictors are said to be highly correlated. In statistics, this phenomenon is called multicollinearity, or in other words **collinearity between variables** (Definition from Wikipedia: Multicollinearity).

Due to collinearity, the coefficient estimates (ß0,ß1,ß2.) ​of the multiple regression may change erratically in response to small changes in the model or the data. But, for a linear regression model to be valid, it is essential that the predictors of the model are linearly independent of each other.

In case of the delivery time dataset, seen previously, the obtained best fit model



shall be valid only if the predictor variables (n.prod and distance) are linearly independent of each other.

The variables are suggested to be linearly dependent if the correlation values are close to -1 or 1.

Hence, to determine the strength of linear relationship between the variables, correlation can be used.

**Variance Inflation Factor**

In addition to correlation, there is another measure called variance inflation factor(VIF) to determine if the predictor variables are independent of each other.

For example, in a given data set, we have target column as 'y', and predictors  'x1', 'x2' and 'x3' . To find out that 'x1', 'x2' and 'x3' are independent of each other, we will calculate VIF for each of the predictors considered. To calculate VIF for the candidate feature 'x1', we need to build a model that predicts 'x1' from the rest of the predictors considered (in this case, 'x2' and 'x3'). Similarly VIF for feature 'x2' and 'x3' will be calculated.

VIFi is given as,

A mathematical equation with numbers and lines

Description automatically generated

where Ri2 is the coeffecient of determination while predicting candidate predictor i, using rest of the candidate predictors.

The range of VIF values start from 1. As a common practice, VIF values can be interpreted as follows -

1 => No correlation between variables

1 to 5 => Slightly correlated

Greater than 5 => Highly correlated

## ****Linear independence of predictors for delivery time dataset****

For the delivery time dataset, the correlation(R) among the variables n.prod and distance is as shown below.

1. *# finding the correlation*
2. np.corrcoef(delivery["n.prod"],delivery["distance"])
3. *# correlation value derived - 0.824215*

Let us now compute the VIF for the independent variables n.prod and distance.

1. from statsmodels.stats.outliers\_influence import variance\_inflation\_factor
2. *#calculating the VIF for each attributes*
3. vif = pd.Series([variance\_inflation\_factor(X.values,idx)
4. for idx in range(X.shape[1])],
5. index=X.columns)
6. print(vif)

A number of numbers on a white background

Description automatically generated

It is observed that the VIF values are greater than 5 for both n.prod and distance features. Also both the features are exhibiting a high correlation value between them. A screenshot of a math problem

Description automatically generated

The coefficient of determination (R­2) for the obtained best fit linear regression model can be determined as shown below.

1. *# score() method for regression model, returns the 'coefficient of determination' (r^2) value for the model*
2. model.score(X,y)
3. *# sample model score as output*
4. *#0.9595937494832257*

It is observed that the R2value for the obtained best fit linear regression model is approximately 0.9595937. The R2value in this case indicates that approximately 95.95 % of the variability observed in the delTime of the vending machine can be explained by variability in the predictor variables, n.prod and distance.

You must be able to recall that the values of R2 ranges between 0 and 1 in Simple Linear Regression. The higher the value of R2 (close to 1), the model is considered as useful.

**Adjusted R-squared**

To establish a best fit linear regression model with minimum error the least squares method is used. For a linear regression model, every additional predictor variable tends to minimize the error of the model. As a result, the R2 value will never decrease for any number of additional predictor variables that is included in the model.

The below code illustrates the increase in the R2for an additional predictor that is included in the model.

1. *# Model with a single predictor - n.prod*
2. model1 = LinearRegression()
3. features = ["n.prod"]
4. target = ["delTime"]
5. model1.fit(delivery[features],delivery[target])
6. print(model1.score(delivery[features],delivery[target]))
7. *#sample model1 score*
8. *#0.9304813135986855*
9. *# Model with multiple predictors - n.prod,distance*
10. model2 = LinearRegression()
11. features = ["n.prod","distance"]
12. target = ["delTime"]
13. model2.fit(delivery[features],delivery[target])
14. print(model2.score(delivery[features],delivery[target]))
15. *#sample model2 score*
16. *#0.9595937494832257*

In other words, the R2 value can be inflated by including more and more predictor variables.

Thus, the use of an additional statistic known as adjusted R2 is suggested. The adjusted R2 takes into account the number of predictor variables and the number of samples or observations included in the Regression model.

The adjusted R2 is defined as:

A black text on a white background

Description automatically generated

Where, **n** is the number of observations and **k** is the number of number of predictor variables in the model.

The adjusted R2 for the obtained best fit model for the delivery time dataset is shown below. Higher the value of Adjusted R2, better is the model. 

1. *#computation of adjusted R-squared*
2. X = delivery[features]
3. y = delivery[target]
4. adjusted\_rscore = 1 - (1-model2.score(X, y))\*(len(y)-1)/(len(y)-X.shape[1]-1)
5. print(adjusted\_rscore)
6. *#sample adjusted R-Squared*
7. *#0.9559204539817008*

**Code:**[Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600395710996480211/web-hosted/assets/MultipleLinearRegression1603435059724.zip) to download the code discussed in Multiple Linear Regression.

**One-hot Encoding for Categorical Variables**

Categorical variables are the variables which take labels as values.

In python, we cannot use categorical predictor variables directly to build a machine learning model. In such cases, categorical values can be encoded into numbers. There are different ways to encode categorical variables. Here, we will understand one of the popular techniques called as One-Hot Encoding.

Consider a dataset of all the properties sold in a city during a certain period of time.

One of the columns in the dataset is 'view'  which describes the the view from the property (eg. lake-view, garden-view, etc.). It can take five values - '0', '1', '2', '3', and '4' as shown below.

Each of the values correspond to the different types of views possible. Even though these values are numbers, they are not quantities. Rather, they are qualititative and hence they should be considered as 'categorical' variables. Hence, we need to encode the variable. We will use One-hot encoding.

Internally, One hot encoding first creates the following mapping –

A table with numbers and symbols

Description automatically generated

From the table it can be seen that it creates as many new columns as there are labels. Wherever the the column and row label matches, it adds a '1'. For all other cells, it adds a '0'.

For the final output, the old column is deleted and the new columns are retained as shown below -A diagram of a number and a number

Description automatically generated with medium confidence

The get\_dummies() of Pandas library can help us get one-hot encoding done easily.

1. *# this is a sample code to demonstrate encoding of the categorical column 'view' using get\_dummies() function*
2. *# this is not complete code and hence not executable.*
3. cat\_features = [ "view"]
4. house\_data\_df = pd.get\_dummies(house\_data\_df,columns=cat\_features)
5. print(house\_data\_df.columns)

A close-up of words

Description automatically generated

## Let us now understand the end-to-end Machine Learning process of Regression analysis.

King County House price dataset needs to be used for this exercise. [Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600395710996480211/web-hosted/assets/housesalesprediction.zip) to download the King County House price dataset. The data represents the details of houses sold in King County, USA in 2014 and 2015. From the given features, predict the price of a house.

1. *#read data from input csv file*
2. house\_data = pd.read\_csv("datasets/kc\_house\_data.csv")

In the following code, we import the data and perform some basic exploratory data analysis.

1. *#Determine the size of the data*
2. print(house\_data.shape)
3. *#output*
4. *#(21613, 21)*
5. *#Determine the columns in the data*
6. print(house\_data.columns)
7. *#View information about the data in each columns*
8. print(house\_data.info())

## Feature Engineering

In the following code, following tasks to be performed:

1. Selecting features to work on. Creating two new features - year and month from the date.
2. Encoding the categorical features
3. Normalizing the continuous numerical features

#### **1. Selecting features**

For this case, let us consider price as the target; and date, bedrooms, bathrooms, sqft\_living, floors, waterfront, view, condition, grade, zipcode as input variables or predictors.

1. house\_data\_df = house\_data[["price","date", "bedrooms",
2. "bathrooms", "sqft\_living", "floors",
3. "waterfront", "view", "condition", "grade","zipcode"]]

Lets split date into year and month to consider that price could depend on year and month of sale due to market conditions

Lets treat bedrooms, bathrooms, floors, waterfront, view, condition, grade, year and month as categorical features.

1. *# Extracting year and month from date*
2. house\_data\_df.loc[:,"year"] = house\_data["date"].str[0:4]
3. house\_data\_df.loc[:,"month"] = house\_data["date"].str[4:6]
4. *#removing date after this extraction*
5. house\_data\_df = house\_data\_df.drop(columns=["date"])

**2. Encoding the categorical features**

When using Pandas get\_dummies() function to encode a categorical column, it is replaced by multiple columns, so that one for each of the values can be taken. For example, if the original 'view' column could take 4 values such as '1', '2', '3' and '4'. After encoding, the 'view' column gets replaced by 4 new columns - 'view\_1', 'view\_2', 'view\_3' and 'view\_4'. Each of these new columns take either a value 0 or 1.

Let us encode the categorical columns - waterfront, view, condition, grade, year, month and zipcode.

1. *# encoding categorical values*
2. cat\_features = ["waterfront", "view", "condition", "grade", "year", "month", "zipcode"]
3. house\_data\_df = pd.get\_dummies(house\_data\_df,columns=cat\_features)
4. print(house\_data\_df.columns)

**3. Normalizing the continuous numerical features**

The features with continuous numerical values are normalized so that, each of these features span across a common range of values, and hence would have equal influence on the model build.

Let us normalize the numerical variables - price, bedrooms, bathrooms, spft\_living and floors.

Note: In the below code,  'StandardScaler' is used for normalizing the data. It works based on the standard deviation and mean value of the numerical columns.

1. from sklearn.preprocessing import StandardScaler
2. *# finding the mean and std deviation of numerical columns*
3. scaler = StandardScaler().fit(house\_data\_df[['price', 'bedrooms', 'bathrooms', 'sqft\_living', 'floors']])
4. *# scaling columns to a common range*
5. house\_data\_normalized = scaler.transform(house\_data\_df[['price', 'bedrooms', 'bathrooms', 'sqft\_living', 'floors']])# Scaled data
6. *#note that output is a 2-Dimensional array of normalized data*
7. house\_data\_normalized

**Replacing the numerical columns with normalized values**

1. *#updating the numerical(normalized) data into the dataframe*
2. house\_data\_df\_normalized=pd.DataFrame(house\_data\_normalized,columns=['price', 'bedrooms', 'bathrooms', 'sqft\_living', 'floors'])
3. house\_data\_df\_normalized=house\_data\_df\_normalized.join(house\_data\_df[house\_data\_df.columns.drop(['price', 'bedrooms', 'bathrooms', 'sqft\_living', 'floors'])])

**Selecting features and target**

1. Y = house\_data\_df\_normalized['price']
2. X = house\_data\_df\_normalized[house\_data\_df\_normalized.columns.drop('price')]
3. print(X.shape)

## ****Splitting into train and test data****

In the below code, we split the data into train and test set in a ratio of 80:20.

1. from sklearn.model\_selection import train\_test\_split
2. X\_train,X\_test,Y\_train,Y\_test = train\_test\_split(X,Y,test\_size=0.2, random\_state=100)
3. X\_train.shape , X\_test.shape
4. *#output*
5. *#((17290, 112), (4323, 112))*

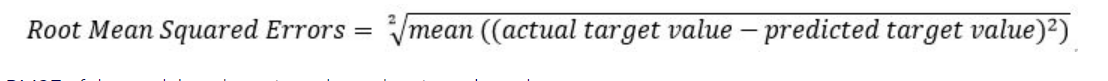
## ****Build a model and evaluate its performance using R-squared****

The required model is built to predict the house price based on the selected predictors.

1. *#importing required module to build the model*
2. from sklearn.linear\_model import LinearRegression
3. *#building and training the model*
4. model = LinearRegression()
5. model.fit(X\_train,Y\_train)
6. *#Evaluating the model on the train and test data for R-Squared score*
7. train\_score=model.score(X\_train,Y\_train)
8. test\_score=model.score(X\_test,Y\_test)
9. print('Train Score (R-Squared): ',train\_score)
10. print('Test Score (R-Squared)',test\_score)
11. *# sample output*
12. *# Train Score (R-Squared): 0.8336987270194174*
13. *# Test Score (R-Squared) 0.8313082620280754*

## Evaluating the model performance using RMSE

Root Mean Squared Error (RMSE) measure can be used to give an estimate of the average error that can be expected from the model.



In the below code, RMSE of the model on the train and test data is evaluated.

1. from sklearn.metrics import mean\_squared\_error
2. *#root mean square error (RMSE) calculation for train data*
3. train\_predictions = model.predict(X\_train)
4. train\_RMSE=mean\_squared\_error(Y\_train,train\_predictions)\*\*0.5
5. *#root mean square error (RMSE) calculation for test data*
6. test\_predictions = model.predict(X\_test)
7. test\_RMSE=mean\_squared\_error(Y\_test,test\_predictions)\*\*0.5
8. print('Train RMSE : ',train\_RMSE)
9. print('Test RMSE : ',test\_RMSE)
10. *# output*
11. *# Train RMSE : 0.4078007927303328*
12. *# Test RMSE : 0.4107102720475509*

RMSE values also confirm previous observation of the model is not overfitting.

**Code :**Download the code for the House Price Prediction problem [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600395710996480211/web-hosted/assets/RegressiononHousePriceData1604236311940.zip).

## ****Prediction accuracy of the regression model****

During the evaluation of a model on train and test data, following are the situations that may be faced:

**Model performance on train and test data is poor (High RMSE)**

Such models are typically referred to as **underfit** models because the model cannot explain the variation in the data reasonably. In such situations, quality and veracity of the data should be relooked and the features selected to build the model should be analysed. This might require more time and effort in engineering the features.

**Model performance on train data is good (Low RMSE, High R-squared) but on test data is poor (High RMSE)**

Such models are typically referred to as **overfit** models, i.e. they have been fit perfectly for the train data but are not generalized enough. In such situations, we may need to:

* Gather more data instances as it is difficult to overfit larger size of data.
* Reduce the complexity of the model - Evaluate which features are important and use only those features to build the model.

Summary

Till now we have seen the typical machine learning process for Regression.

Note that even though this process was discussed in the context of multiple linear regression, this is a generic process that can be applied to simple linear regression, multiple linear regression, and other classification algorithms as well.

A diagram of a machine learning process

Description automatically generated

Exercise:

Use the Boston dataset (Click [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600397032718336230/web-hosted/assets/bostonhousing1603883314333.zip) to download the Boston dataset) and perform the following activities:

1.    Consider the columns, ‘RM’, ‘DIS’, ‘TAX’, ‘INDUS’ as predictors, and ‘MEDV’ as the target variable

2.    Calculate the Variance Inflation Factor for each of the selected predictors. Based on the VIF factor, finalize the list of predictors.

3.    For every predictor identified, visualize its association with the target column using scatter plot.

4.    Split the data into train and test datasets, in the ratio of 67:33.

5.    Build a Linear Regression model, to predict the target variable using the selected predictors.

6.    Observe the coefficients and intercept values for the model.

7.    Evaluate the model using mean squared error values, R-squared values, and adjusted R-Squared values, on the train and the test data.

A screenshot of a computer screen

Description automatically generated

Ans: Model1

Q3 of 3outlined\_flag

Suppose, you want to build a regression model to predict the price of a car from its age and miles driven. You find that the VIF for both the predictors is nearly 9. Is it advisable to build a linear regression model with both the predictors?

Ans: no

**Introduction to Classification**

## Classification

Classification is a supervised Machine Learning technique that helps in predicting categorical or discrete (non-continuous) outputs. Classification algorithms can be used to categorise/classify instances or samples. For example, whether a  mail is a spam or a transaction is fraudulent.

Classification algorithms learn to map the input data to a category or a class from the training data.

Logistic Regression is a supervised Machine Learning algorithm, primarily used for binary classification.  For a given sample, Logistic Regression computes the probability of a sample belonging to each of the classes.

The probabilities are computed using the non-linear sigmoid or logistic function given below:

A math equations on a white background

Description automatically generated

Here,

x1, x2​​​​, x3,...,xn​​​ are the input feature values or predictor values.

ß1, ß2, ..., ßn are coefficients associated with the input features which help decide how much influence a feature will have on the final sigmoid(z) value.

ß0 is called the intercept.

The graph of the sigmoid function is given below –

A graph of a function

Description automatically generated

# Building a Logistic Regression model for Coronary heart disease:

To build the model, divide the entire set of samples that we have into training data and test data. Training data is used by the algorithm to learn the model. Test data is used to validate the model.

1. *# Importing the required class*
2. from sklearn.model\_selection import train\_test\_split
3. *# Specifying the columns as predictor and target variable*
4. predictors = ["age"]
5. target = "chd"
6. X = chd[predictors] *# X contains the predictor values or input data*
7. y = chd[target] *# y contains the target labels*
8. *# Spliting the data in training and test set in 70:30 ratio*
9. X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=0)
10. *# Checking the shapes of the resulting datasets*
11. print("Shape of X\_train:", X\_train.shape)
12. print("Shape of y\_train:", y\_train.shape)
13. print("Shape of X\_test:", X\_test.shape)
14. print("Shape of y\_test:", y\_test.shape)

From the ouput you can figure out that setting the test\_size parameter to 0.3 splits the data such that 30% of all samples is set aside for test dataset. (The random\_state is set to 0 so that when you run the program, you get the same resulting datasets.)

Now the model can be built using sklearn.linear\_model.LogisticRegression class as shown below.

1. *# Importing the required class*
2. from sklearn.linear\_model import LogisticRegression
3. *# Creating the object of the class LogisticRegression*
4. model = LogisticRegression()
5. *# Training the model using the training data*
6. model.fit(X\_train,y\_train)
7. *# Getting the intercept and the coefficients of the model*
8. print("Intercept:",model.intercept\_,"\nCoefficients:", model.coef\_)

Using this logistic model, the probability of occurrence of coronary heart disease for a person aged 29 can be predicted as shown below:

1. *# Creating a sample data*
2. test=np.array([29]).reshape(1,-1)
3. *# Predicting the probabilities for each of the class labels*
4. print("Predicted probability for class '0' and '1' respectively:", model.predict\_proba(test))
5. *# Predicting the final class label or target value*
6. print("\nPredicted target i.e. 'chd' value:",model.predict(test))

The predicted probability for class '0' is higher than predicted probability of class '1' . So class '0' is chosen as the final target value for this instance.

Similarly, the probabilities for the class labels can be predicted for other samples.

Let us evaluate the model's performance on training and test data using 'accuracy' measure.

**Accuracy = no. of correct predictions / total predictions**

The score() function of sklearn gives the accuracy for classification models.

1. print("Accuracy score of the model on training data:", model.score(X\_train, y\_train))
2. print("Accuracy score of the model on test data:", model.score(X\_test, y\_test))

The output shows that model is 70% accuracte in prediction. It can be assumed that the model is not overfitting to the training data because the accuracy on test data is similar to that of the training data

**Code:**[Click here to download](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600400530505728236/web-hosted/assets/LogisticRegressionCHD1603525250845.zip) the code for Logistic Regression model for Coronary heart disease.

# Logistic Regression on Credit Risk data:

Now that you are familiar with the basic concept behind logistic regression,let us understand the use of logistic regression on another problem by building  a model to classifying the credit risk for a loan applicant.

The first step is to import basic libraries, data and understanding the data.

Download the Credit risk dataset [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600400530505728236/web-hosted/assets/credit_risk.csv).

1. *# Importing required libraries*
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. %matplotlib inline
6. *# Importing the dataset*
7. credit\_data = pd.read\_csv("datasets/credit\_risk.csv")

The data is too large to fit in one screen. So let us see its information below.

You can see that there are a total of 1000 samples or records of past loan applications. Each record is defined by 21 attributes or column values. The target column of interest is the 'class' column.

1. *# Understanding the values the 'class' column (our target column in this analysis) can take*
2. credit\_data['class'].unique()

So, the target column 'class' can take two values -- 'good' and 'bad' which states whether the past loan application was a good or bad credit risk.

From the output of info() function it can be seen that there are no Null values in any of the columns (since every column has the same number of non-null values as the number of rows). However, some of the predictors have categorical values (given by the datatype, 'object'), which need to be encoded into numbers. Note that, target column doesn't need encoding when it has categorical values.

**Splitting Credit Risk data into Training and testing data**

In this example, the data is split into training and test datasets in the ratio of 85:15.

1. *# Importing the required module*
2. from sklearn.model\_selection import train\_test\_split
3. *#splitting data into train and test datasets in 85:15 ratio*
4. X\_train,X\_test,y\_train,y\_test = train\_test\_split(credit\_data\_encoded, y,test\_size=0.15,random\_state=100)
5. *# Checking the shapes of the resulting datasets*
6. print("Shape of X\_train:", X\_train.shape)
7. print("Shape of y\_train:", y\_train.shape)
8. print("Shape of X\_test:", X\_test.shape)
9. print("Shape of y\_test:", y\_test.shape)

**Building the model**

Let us build the logistic regression model using sklearn.

1. *# Importing the required class.*
2. from sklearn.linear\_model import LogisticRegression
3. *# Instantiating the required algorithm for model building.*
4. model = LogisticRegression()
5. *# Building the model based on the training data.*
6. model.fit(X\_train,y\_train)

Now that you have built the model, the next step is to evaluate the model's performance or accuracy on the training and test data:

1. *# Getting the accuracy on training data*
2. train\_accuracy = model.score(X\_train,y\_train)
3. print("Train accuracy = ", train\_accuracy)
4. *# Getting the accuracy on test data*
5. test\_accuracy = model.score(X\_test,y\_test)
6. print("Test accuracy = ", test\_accuracy)

The finding shows that the accuracy on the test data is similar to the training data. Therefore, it can be assumed that the model is not overfitting the training data.

**Measuring Model Performance using Confusion Matrix**

Confusion matrix helps in assessing how good a model is by comparing the actual target values with the predicted target values.

Let us see how to generate the Confusion Matrix for a model in sklearn:

1. *# Predicting targets based on the model built*
2. train\_predictions = model.predict(X\_train)
3. test\_predictions = model.predict(X\_test)
4. *# Importing the required function*
5. from sklearn.metrics import confusion\_matrix
6. *# Creating a confusion matrix on the training data*
7. train\_conf\_matrix = confusion\_matrix(y\_train,train\_predictions)
8. *# Converting the train\_conf\_matrix into a DataFrame for better readability*
9. pd.DataFrame(train\_conf\_matrix,columns=model.classes\_,index=model.classes\_)

A white background with black text

Description automatically generated

The rows of a Confusion Matrix represent the actual target values and the columns represent the predicted target values.

In the above matrix for training data, we can observe that the model predicted -

* 140 actually 'bad' credit risks as 'bad'
* 117 actually 'bad' credit risks as 'good'
* 60 actually 'good' credit risks as 'bad'
* 533 actually 'good' credit risks as 'good'

1. *# Confusion matrix for the test data*
2. test\_conf\_matrix = confusion\_matrix(y\_test,test\_predictions)
3. pd.DataFrame(test\_conf\_matrix,columns=model.classes\_,index=model.classes\_)

A screenshot of a computer

Description automatically generated

In the above matrix for test data, we can see that the model predicted -

* 18 actually 'bad' credit risks as 'bad'
* 25 actually 'bad' credit risks as 'good'
* 15 actually 'good' credit risks as 'bad'
* 92 actually 'good' credit risks as 'good'

'Accuracy' can be computed from Confusion Matrix as well using the following expression:

Let us compute the accuracy for our training and test datasets using the above expression.

A black text on a white background

Description automatically generated

1. *# Calculating train accuracy from confusion matrix*
2. train\_correct\_predictions = train\_conf\_matrix[0][0]+train\_conf\_matrix[1][1]
3. train\_total\_predictions = train\_conf\_matrix.sum()
4. train\_accuracy = train\_correct\_predictions/train\_total\_predictions
5. print(train\_accuracy)

1. *# Calculating test accuracy from confusion matrix*
2. test\_correct\_predictions = test\_conf\_matrix[0][0]+test\_conf\_matrix[1][1]
3. total\_predictions = test\_conf\_matrix.sum()
4. test\_accuracy = test\_correct\_predictions/total\_predictions
5. print(test\_accuracy)

A screenshot of a computer

Description automatically generatedWe can also get the Precision, Recall and F1-scores using sklearn as shown below -

1. *# Importing the required function*
2. from sklearn.metrics import classification\_report
3. *# Generating the report and printing the same*
4. print(classification\_report(y\_test,test\_predictions))

A screenshot of a computer

Description automatically generated

We can also get the Precision, Recall and F1-scores using sklearn as shown below -

1. *# Importing the required function*
2. from sklearn.metrics import classification\_report
3. *# Generating the report and printing the same*
4. print(classification\_report(y\_test,test\_predictions))

A screenshot of a computer screen

Description automatically generated

**Code:**[Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600400530505728236/web-hosted/assets/LogisticRegressionCreditriskPrediction1603526803544.zip) to download the code for credit risk prediction using logistic regression.

# Building a Logistic Regression model for Coronary heart disease:

To build the model, divide the entire set of samples that we have into training data and test data. Training data is used by the algorithm to learn the model. Test data is used to validate the model.

1. *# Importing the required class*
2. from sklearn.model\_selection import train\_test\_split
3. *# Specifying the columns as predictor and target variable*
4. predictors = ["age"]
5. target = "chd"
6. X = chd[predictors] *# X contains the predictor values or input data*
7. y = chd[target] *# y contains the target labels*
8. *# Spliting the data in training and test set in 70:30 ratio*
9. X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=0)
10. *# Checking the shapes of the resulting datasets*
11. print("Shape of X\_train:", X\_train.shape)
12. print("Shape of y\_train:", y\_train.shape)
13. print("Shape of X\_test:", X\_test.shape)
14. print("Shape of y\_test:", y\_test.shape)

From the ouput you can figure out that setting the test\_size parameter to 0.3 splits the data such that 30% of all samples is set aside for test dataset. (The random\_state is set to 0 so that when you run the program, you get the same resulting datasets.)

Now the model can be built using sklearn.linear\_model.LogisticRegression class as shown below.

1. *# Importing the required class*
2. from sklearn.linear\_model import LogisticRegression
3. *# Creating the object of the class LogisticRegression*
4. model = LogisticRegression()
5. *# Training the model using the training data*
6. model.fit(X\_train,y\_train)
7. *# Getting the intercept and the coefficients of the model*
8. print("Intercept:",model.intercept\_,"\nCoefficients:", model.coef\_)

Using this logistic model, the probability of occurrence of coronary heart disease for a person aged 29 can be predicted as shown below:

1. *# Creating a sample data*
2. test=np.array([29]).reshape(1,-1)
3. *# Predicting the probabilities for each of the class labels*
4. print("Predicted probability for class '0' and '1' respectively:", model.predict\_proba(test))
5. *# Predicting the final class label or target value*
6. print("\nPredicted target i.e. 'chd' value:",model.predict(test))

The predicted probability for class '0' is higher than predicted probability of class '1' . So class '0' is chosen as the final target value for this instance.

Similarly, the probabilities for the class labels can be predicted for other samples.

Let us evaluate the model's performance on training and test data using 'accuracy' measure.

**Accuracy = no. of correct predictions / total predictions**

The score() function of sklearn gives the accuracy for classification models.

1. print("Accuracy score of the model on training data:", model.score(X\_train, y\_train))
2. print("Accuracy score of the model on test data:", model.score(X\_test, y\_test))

The output shows that model is 70% accuracte in prediction. It can be assumed that the model is not overfitting to the training data because the accuracy on test data is similar to that of the training data

**Code:**[Click here to download](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600400530505728236/web-hosted/assets/LogisticRegressionCHD1603525250845.zip) the code for Logistic Regression model for Coronary heart disease.

**Exercise 1:**

Perform the following activities using Bank Marketing dataset.

1. Consider duration, age, and campaign columns as predictors and the column “y”(states whether the client will subscribe a term deposit or not) as target  variable
2. Use 70% of the data as training data set and 30% of data as testing data set.
3. Build a logistic regression model.
4. Determine the classification accuracy rate of the model on both train and test set.

The Bank Marketing dataset can be downloaded [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600399366586368181/web-hosted/assets/bankadditionalfull1603957400921.zip). It is taken from UCI Machine Learning Repository - [[Moro et al., 2014] S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems, Elsevier, 62:22-31, June 2014]

**Exercise 2:**

Perform the following activities using banknote authentication dataset.

1. Consider 'variance', 'skewness', 'curtosis', 'entropy' columns as predictors and the column 'class' as the target  variable
2. Build a logistic regression model.
3. Calculate the classification accuracy, precision and recall for the model

The banknote authentication data set can be downloaded [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600399366586368181/web-hosted/assets/databanknoteauthentication1603957797786.zip). It is taken from UCI Machine Learning Repository.

## Decision tree algorithms

## Attribute selection measures (7/12)

Attribute selection measures compare different predictor attributes and rank them for the purpose of model building. These measures convey information about the relevance of a predictor when compared to other predictors. Decision tree algorithms makes use of the ranked list given by the attribute selection measures and then selects the best attribute to split the dataset.

Three of the most commonly used attribute selection measures to induct a decision tree are:

* Information gain
* Gain ratio
* Gini Index

Let us delve deeper and understand how to build trees using the classes in **sklearn.tree** package. The attribute selection measures used while building trees are either Gini index or Information gain.

## Decision tree on credit risk data set

By now you must be familiar with the basic principle of Decision Trees. Let us now use decision trees to classify if a loan applicant is a good or bad credit risk using the Credit Risk dataset.

[Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600402391597056241/web-hosted/assets/creditriskdataset1603523640815.zip) to download the credit\_risk dataset.

The first step is to load the data from csv file and understand the data.

1. *# Importing required libraries*
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. %matplotlib inline
6. *# Reading data from input csv file*
7. credit\_data = pd.read\_csv("datasets/credit\_risk.csv")
8. *# Uncomment the following line to view the sample data*
9. *# credit\_data.head()*

1. *# Understanding the various attributes*
2. credit\_data.info()

You can see that there is a total of 1000 instances or samples. Each column has 1000 non-null values. In other words, there are no null values that need imputing. Also, target column of interest is the "class" column which takes two values - 'good' and 'bad'.

After loading the data, the next step is to select relevant features, perform necessary Feature Engineering and then split the data into train and test sets.

**Setting the predictors and target:**

Consider all the attributes except for the target column, "class" as potential predictors for building the tree.

1. *# Selecting the predictor attributes*
2. X = credit\_data.columns.drop("class")
3. *# Selecting the target*
4. y = credit\_data['class']

**Encoding the categorical values:**

Encode the categorical data so that it can be used to build a model using sklearn.

1. *# Encoding all the predictor variables to convert the categorical values to numerical values.*
2. credit\_data\_encoded = pd.get\_dummies(credit\_data[X])
3. print("Total number of predictors after encoding = ", len(credit\_data\_encoded.columns))
4. *# Printing the list of columns after encoding to understand the encoding process*
5. credit\_data\_encoded.columns

After encoding, there are 61 predictor attributes. Each of the non-numerical columns has been replaced by new columns as the number of its values. The newly generated columns take either 0 or 1 as the value.

**Splitting the data into train and test sets:**

Now let us split the data into train and test dataset in the ratio 85:15.

1. *# Import the required function*
2. from sklearn.model\_selection import train\_test\_split
3. *#splitting data into train and test datasets*
4. X\_train,X\_test,y\_train,y\_test = train\_test\_split(credit\_data\_encoded, y,test\_size=0.15,random\_state=100)
5. *# Printing the shape of the resulting datasets*
6. print("Shape of X\_train and y\_train are:", X\_train.shape, "and", y\_train.shape, " respectively")
7. print("Shape of X\_test and y\_test are:", X\_test.shape, "and", y\_test.shape, " respectively")

Using the same random state value will ensure that you get the same results everytime.

**Building the model using Scikit-Learn:**

Now let us build the decision tree model using sklearn.tree.DecisionTreeClassifier.

1. *# Importing required class*
2. from sklearn.tree import DecisionTreeClassifier
3. *# Creating an object of the DecisionTreeClassifier model*
4. model = DecisionTreeClassifier(random\_state = 1)
5. *# Training model on the training data*
6. model.fit(X\_train,y\_train)

You can observe the way to create a Decision Tree model is similar to creating a Logistic Regression model. This is an advantage of using the sklearn package. It provides a simple interface for building machine learning models.

The output of the execution of the above lines of code shows the different parameters that defines the built decision tree model. These parameters are known as hyper-parameters of a model. The values of these hyperparameters can be set explicitly to get different models.

You can learn more about the decision tree classifier hyperparameters from the [sklearn documentation page](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html" \t "_blank).

To use the model for predicting the label of a data, you can use the predict() as shown below.

1. *# Predicting target values using the model built on training data*
2. train\_predictions = model.predict(X\_train)
3. test\_predictions = model.predict(X\_test)

**Visualizing the decision tree:**

You can visualize the inducted decision tree using the export\_graphviz() function of the sklearn.tree as shown below.

1. *# Importing the required libraries (Ensure that they are already installed.)*
2. from sklearn.tree import export\_graphviz
3. import graphviz
4. *# Generating the tree*
5. dot\_data = export\_graphviz(model, out\_file=None,
6. feature\_names=credit\_data\_encoded.columns,
7. class\_names=model.classes\_,
8. )
9. graph = graphviz.Source(dot\_data)
10. graph

Only a part of the generated tree is shown below. Practice running the code to see the complete tree.

A diagram of a root system

Description automatically generated

# Exercise

For the Bank Marketing dataset available in UCI Machine Learning Repository, induct a decision tree to predict whether a client will subscribe to a term deposit or not (given in column 'y').

* Consider the column 'y' as the target variable. From the remaining columns, consider all except 'duration' as predictors.
* Encode the predictors to convert them to numerical data.
* Split the data into train and test with a ratio of 80:20.
* Build a model using Decision Tree algorithm, to predict the target.
* Determine the accuracy, precision and recall for every class in the outcome.
* Modify the hyperparameters, and build a new model. Compare the performance of both models.

Click [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600400291553280199/web-hosted/assets/bankadditionalfull1603958996760.zip) to download the required data.

KNN

## Classification using k-NN:

The k-Nearest Neighbors(kNN) algorithm determines the target value of a new data point or instance by comparing it with existing data points or instances that are closest to it. The target values of the k-closest instances are aggregated and taken as the target output for the new data point. Let us understand the algorithm with the help of the 'defaulter' dataset. It contains data about customers defaulting on loans. [Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600402331557888183/web-hosted/assets/defaulter1604733431952.zip) to download the dataset used here.

*#reading data from input csv file*

defaulter = pd.read\_csv("datasets/defaulter.csv")

defaulter

*#visualizing data using seaborn - pariplot*

import seaborn as sns

sns.pairplot(defaulter,hue="defaulter",

x\_vars="income",y\_vars="balance",height=4)

You can see a customer with a balance of 1000 and income of 16,900 is currently not classified as either defaulter/non-defaulter. We can use kNN to classify this instance.

A graph of a number of people

Description automatically generated with medium confidence

**Euclidean Distance**

The kNN algorithm identifies the nearest neighbors based on Euclidean distance\* - a commonly used distance metric. The Euclidean distance between two (tuples) - X1 (x11, x12, x13, .. x1n) and X2 (x21, x22, x23, .. x2n) can be computed as:

A black rectangular object with a square and a square with a square and a square with a square with a square and a square with a square with a square and a square with a square with a

Description automatically generated

where, x11, x12, x13, .. x1nare the numeric attributes of X1and x21, x22, x23, .. x2nare the numeric attributes of X2.

In the given scenario, each instance has two numeric attributes such as balance and income. Thus, each instance here is represented as a point in a 2-dimensional space.

Let us now compute the Euclidean distance between a few tuples in the given scenario.

Let X1 be the tuple at index 0 (balance=817.1804, income=12106.14, defaulter=No) and X2 be the tuple at index 1 (balance=1486.998, income=17854.4, defaulter=Yes).

The Euclidean distance between X1 and X2 can be computed as:

1. *# retrieving first data point from the dataset*
2. x1 = defaulter.loc[0,["balance","income"]]
3. *# retrieving second data point from the dataset*
4. x2 = defaulter.loc[1,["balance","income"]]
5. *# Euclidean distance between first and second data point*
6. np.linalg.norm(x1-x2)
7. *#output*
8. *#5787.1537602581275*

\**Note: Euclidean distance is used as a distance metric when the data tuple comprises of numeric attributes. Distance metrics such as Hamming distance can be used when the data tuple comprises of categorical attributes. This course discusses kNN based on Euclidean distance only.*

3/9 :

## ****Limitation of Euclidean Distance****

One of the limitations of Euclidean distance is that attributes with larger ranges contribute more value to the Euclidean distance.

For example, if the numeric attributes are age and loan amount, then it can be observed that the typical age in years for humans may range from 0 to 100 years, however, the loan amount in dollars may range from 0 dollars to several thousand dollars.

Let us illustrate this with a simple example. Consider 3 data tuples T1(age=26, loanAmt=1000), T2(age=66, loanAmt=1000) and T3(age=36, loanAmt=10000).

The Euclidean distance of the tuple T3 from T1 and T2 can be computed as:

1. *# considering 3 random datapoints - t1, t2, t3*
2. t1 = np.array([26,1000])
3. t2 = np.array([66,1000])
4. t3 = np.array([36,10000])
5. *# distance between t1 & t3*
6. dist\_t3\_t1 = np.linalg.norm(t3-t1)
7. *# distance between t2 & t3*
8. dist\_t2\_t3 = np.linalg.norm(t2-t3)
9. print("Distance between t2 and t3",dist\_t2\_t3,
10. "\nDistance between t1 and t3",dist\_t3\_t1)
11. *#output*
12. *#Distance between t2 and t3 9000.049999861112*
13. *#Distance between t1 and t3 9000.00555555384*

You can observe that the difference in the values of age for T1 and T3 (i.e. 26-36=10) is significantly less than the difference in the values of age for T2 and T3 (i.e. 66-36=30). However, the Euclidean distance between the tuples T1 and T3 is same as that of the Euclidean distance between the tuples T2 and T3, suggesting that the parameter loanAmt has a high influence on the Euclidean distance as compared to the age.

To avoid this situation, all the numeric attributes of the tuples can be **normalized** before they are used for computing the Euclidean distance. Normalization is essential to take into consideration, the different measurement scales of the attributes. Normalization is not just applied to kNN, it is often considered as a good practice to normalize the data before running any machine learning algorithm.

## 4/9 Normalizing the Data

There are several normalization techniques that can be applied. These methods are available in the package 'sklearn.preprocessing'.

Let us now normalize the balance and income in the defaulter dataset and find the euclidean distance between instances X1 (tuple at index 0 - balance=817.1804, income=12106.14, defaulter=No) and X2(tuple at index 1 - balance=1486.998, income=17854.4, defaulter=Yes).

To normalize the data, use the min-max normalization. This normalization transforms the value *v* of a numeric attribute A to a value *v'*, where

We shall first normalize the values of the attributes of X1 and X2. Let us use the min-max normalization technique to normalize the 'balance' and 'income'.

The below code demonstrates the use of MinMaxScaler in sklearn.preprocessing to normalize the data.

1. *# Using MinMaxScaler of Scikit Learn library for Normalization of all features*
2. from sklearn.preprocessing import MinMaxScaler
3. scaler = MinMaxScaler()
4. scaled\_values = scaler.fit\_transform(defaulter[["balance","income"]])
5. defaulter["norm\_balance"] = scaled\_values[:,0]
6. defaulter["norm\_income"] = scaled\_values[:,1]
7. defaulter

# 5/9 Prediction based on 'k' Nearest Neighbors

The Euclidean distance of the new normalized tuple X11'(tuple with index 10) from all other tuples is sorted in ascending order as shown below:

1. *# Lambda function to calculate Euclidean distance*
2. x11 = defaulter.loc[10,["norm\_balance","norm\_income"]]
3. dist\_to\_X11 = lambda x: np.linalg.norm(x-x11)
4. *# calculating Euclidean Distance*
5. defaulter["dist\_toX11"] = defaulter[["norm\_balance",
6. "norm\_income"]].apply(dist\_to\_X11,axis=1)
7. *# sorting the datapoints based on distance to find the 'k' nearest neighbors*
8. defaulter.sort\_values("dist\_toX11")

The Euclidean distance between normalized tuples X1' and X2' is

1. *#distance between two normalized data points*
2. x1\_n = defaulter.loc[0,["norm\_balance","norm\_income"]]
3. x2\_n = defaulter.loc[1,["norm\_balance","norm\_income"]]
4. np.linalg.norm(x1\_n-x2\_n)
5. *#output*
6. *#1.1489497815843785*

 Recall that the kNN algorithm identifies the k nearest neighbors of a new tuple and assigns it a class label based on the class labels of its nearest neighbors.

Let us consider the value of k as 1, which means that, the kNN algorithm assigns a class label based on its nearest neighbor. The kNN algorithm would therefore classify the tuple X11 as default= Yes based on its nearest neighbor X3 (tuple at index 2) since the nearest neighbor (X3) has a class label of default=Yes.

6/9

For all values of k greater than 1, a data tuple is classified by a majority vote of its neighbors. The tuple will be assigned the class that is most common among its k nearest neighbors.

If the value of k was selected as 3, the 3-nearest-neighbors of the data tuple X11 (balance=1000, income=16900) are identified. In this case they are X3 (balance=1102.435, income=17391.78, defaulter=Yes), X9(balance=903.1768, income=15810.57, defaulter=No) and X8(balance=870.4124, income=17685.45, defaulter=No) based on the Euclidean distances computed earlier.

The new tuple would be classified as defaulter/non-defaulter based on the majority vote of the neighbors X3, X9and X8. In this case, the majority vote indicates that the new tuple (X11) would be a non-defaulter.

For k=2, X3 (balance=1102.435, income=17391.78, defaulter=Yes) and X9(balance=903.1768, income=15810.57, defaulter=No) are the 2 closest neighbors of X11 based on the Euclidean distances. In this case, you can see a tie in the votes (i.e. 1 vote for defaulter and 1 vote for non-defaulter).

Therefore, it is difficult to classify the data tuple X11 as defaulter or a non-defaulter because the vote count results in a tie. Typically for **binary-class** cases (like defaulter= Y/N), an odd value of k is preferred to avoid such tie situations. For **multi-class** cases (more than 2 classes), the ties can be broken by assigning a class at random or by assigning the class that occurs most frequently.

# 7/9 Building kNN model using sklearn

Getting to know the working of the algorithm , let us now implement it on a defaulter dataset (with 10000 data instances) while following the machine learning process outlined earlier.

Click [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600402331557888183/web-hosted/assets/defaulter1603944816113.zip) to download the dataset.

## Step 1: Loading the data

1. *#reading in data from csv file*
2. defaulter = pd.read\_csv("datasets/defaulter.csv")

# Step 2: Feature engineering - normalization

Here you can normalize the balance and income columns in the data.

1. from sklearn.preprocessing import MinMaxScaler
2. *#applying MinMaxScaler to the default dataset*
3. scaler = MinMaxScaler()
4. features\_to\_scale = ["balance","income"]
5. scaled\_values = scaler.fit\_transform(defaulter[features\_to\_scale])
6. defaulter["norm\_balance"] = scaled\_values[:,0]
7. defaulter["norm\_income"] = scaled\_values[:,1]
8. defaulter.head()

8/9

# Step 3: Spliting the dataset into train and test data

1. from sklearn.model\_selection import train\_test\_split
2. *# selecting the predictors and target*
3. X=defaulter[['norm\_balance','norm\_income']]
4. Y=defaulter['defaulter']
5. *# splitting data into train and test data set*
6. X\_train,X\_test,Y\_train,Y\_test = train\_test\_split(X,Y,test\_size=0.2,random\_state=100)

# Step 4: Building the model

1. from sklearn.neighbors import KNeighborsClassifier
2. *#kNN with k=3*
3. model = KNeighborsClassifier(n\_neighbors=3,metric="euclidean")
4. model.fit(X\_train,Y\_train)

# 9//9 Step 5: Evaluate model performance on train and test sets

1. train\_accuracy = model.score(X\_train,Y\_train)
2. test\_accuracy = model.score(X\_test,Y\_test)
3. print(train\_accuracy,test\_accuracy)
4. *#output*
5. *#0.977375 0.967*

In our example, value chosen for k is 3 while building the model. However, the choice of k affects the performance of the model. So, for the kNN algorithm, k is a hyperparameter that needs to be tuned.

The below code demonstrates the train and test accuracy for different values of k on the defaulter dataset:

1. train\_accuracies = []
2. test\_accuracies = []
3. *# building KNN model for values of k from 1 to 100*
4. k\_vals = [i for i in range(1,100)]
5. features = ["norm\_balance","norm\_income"]
6. target = "defaulter"
7. for k in k\_vals:
8. model = KNeighborsClassifier(n\_neighbors=k,metric='euclidean')
9. model.fit(X\_train,Y\_train)
10. train\_accuracy\_k = model.score(X\_train,Y\_train)
11. test\_accuracy\_k = model.score(X\_test,Y\_test)
12. train\_accuracies.append(train\_accuracy\_k)
13. test\_accuracies.append(test\_accuracy\_k)
14. plt.plot(k\_vals,train\_accuracies)
15. plt.plot(k\_vals,test\_accuracies)

A graph of a train and test

Description automatically generated

**Code:**[Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600402331557888183/web-hosted/assets/kNearestNeighbours1603463455432.zip) to download the code discussed in K-Nearest Neighbors.

# Exercise

Perform the following activities using Bank Marketing dataset.

1. Consider the columns - duration, and age as predictors and the column “y”(indicates whether the client will subscribe for a term deposit) as the target variable
2. Use 60% of the data as training dataset and 40% of data as testing dataset.
3. Build a kNN classification model.
4. Plot the classification accuracies on train and test data for different values of k.

The Bank Marketing data set dataset can be downloaded [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600395220877312180/web-hosted/assets/bankadditionalfull1603960061414.zip).

It is taken from UCI Machine Learning Repository - [[Moro et al., 2014] S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems, Elsevier, 62:22-31, June 2014]

Support Vector Machine

## 1/5 ****Introduction to Support Vector Machine (SVM)****

Support Vector Machines (SVM) is another classification algorithm that classifies data into one category or the other by using hyperplanes.

Example: If you want to classify an iris flower as setosa or non-setosa based on petal width and petal length, we could use a separating line to do the same as shown below.

A diagram of a graph

Description automatically generated

Note: the separating line here has been drawn manually for the purpose of illustration.

Consider the following plot which illustrates a few lines that can separate data into setosa or non-setosa.

A diagram of a line and line graph

Description automatically generated

## 2/5 Choosing the optimal hyperplane

In the plot shown below, you can choose 3 parallel lines that divide the data.

A diagram of lines and dots

Description automatically generated

Note: The lines and the additional grey points in the above plot have been manually inserted for the purpose of illustration.

You can observe that all 3 lines drawn here can separate the given data. Line 3 is closest to non-setosa and Line 1 is closest to setosa. These lines (Line 1 and 3) are called as margin lines as they represent the boundary of a category. Consequently, the line that is midway (Line 2) between the margin lines would be the optimal line.

When the number of predictors is 3, you get an optimal plane. In general, for n features (represented in n-dimensional space), you can find the optimal hyperplane.

SVM essentially creates a model such that it finds the widest possible margins and thus the optimal hyperplane.

When data is not separable by a line or a plane, SVM maps the data into a higher dimensional space, where it can be separated using a linear hyperplane. The mapping function used to transform the data into the higher dimensional space is called 'kernel' function.

Note: The lines and the additional grey points in the above plot have been manually inserted for the purpose of illustration.

From the above plot, you can observe that all three lines – Line 1, 2 and 3 can separate the given sample data points into the two classes (setosa and non-setosa). Points on the left of these lines represent setosa flowers and the ones on the right represent the non-setosa flowers.

Let us now consider three new data points (A, B and C) where the class labels are unknown. We can observe that while Line 3 classifies A as setosa, Line 1 and 2 classify it as non-setosa.

Thus, you can conclude that:

1. There can be many lines that can separate the known data correctly.
2. These lines may agree/disagree with each other when new data points are presented.
3. Point A lies very close to Line 2 and 3. It is possible that the lines would have misclassified point A.

Based on the above observations, the goal of SVM can be redefined to finding an **optimal hyperplane** that separates the data into classes.

## 3/5 ****Building an SVM model****

 You can look at a demo on building an SVM model.

#### **Reading data**

Consider the Iris dataset which provides measurements of sepal length, sepal width, petal length, and petal width for 50 flowers from each of 3 species (setosa, versicolor, and virginica) of Iris. Click [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600398349197312188/web-hosted/assets/iris.csv) to download the iris dataset.

First, you will see how to build a binary classifier using SVM, that will help us identify whether a given instance of flower is of the species 'versicolor'.

1. *#reading input from csv file*
2. iris\_data = pd.read\_csv("datasets/iris.csv")
3. iris\_data.head()

**Feature Engineering**

Create a new column in the dataframe (v\_nv), that distinguishes the species - 'versicolor'(marked by 0) from rest. Then, build binary classifier based on this new column.

1. *#creating new column 'v\_nv', to distinguish versicolor species from rest*
2. *#the below lambda function returns 0 for 'versicolor' species and returns 1 for rest.*
3. v\_nv\_fn = lambda x: 0 if x=="versicolor" else 1
4. *# new column added into dataframe*
5. iris\_data["v\_nv"] = iris\_data["Species"].apply(v\_nv\_fn)
6. iris\_data[iris\_data['v\_nv']==0].head()
7. *#visualization using seaborn - pairplot*
8. import seaborn as sns
9. sns.pairplot(iris\_data,
10. x\_vars = "Petal.Length",y\_vars="Petal.Width",
11. hue="v\_nv",height=5)

In the above plot, you can find the separating hyperplane to classify data as versicolor and non-versicolor. You can observe that the data is not linearly separable i.e, it cannot be separated using a straight line.

A graph of a number of dots

Description automatically generated

4/5 **Model Creation**

Now let us build a model based on Support Vector Classification, to predict if a data instance is of the species 'versicolor'.

1. *#Support Vector Classification*
2. from sklearn.svm import SVC
3. *#setting predictors and target*
4. X = iris\_data[["Petal.Length","Petal.Width"]]
5. Y = iris\_data["v\_nv"]
6. *# model building*
7. model = SVC()
8. model.fit(X,Y)
9. model.score(X,Y)
10. *# 0.9533333333333334*

**Model Visualization**

A separating hyperplane that divides the iris dataset into versicolor and non-versicolor categories, looks as shown below.

Note:'mlxtend' is another machine learning library and provides various useful tools for data science applications.

Here, plot\_decision\_regions() function of mlxtend library is used for plotting the decision regions.

1. from mlxtend.plotting import plot\_decision\_regions
2. features = np.array(X)
3. target = np.array(Y).ravel()
4. plot\_decision\_regions(features,target,clf=model)
5. plt.xlabel("Petal length")
6. plt.ylabel("Petal width")
7. plt.title('Decision boundary of SVM on iris data')

A diagram of a blue and orange circle

Description automatically generated with medium confidence

## 5/5 ****Multi-class classification****

Further, even though SVM is considered as a binary classifier, it can be used for multi-class classification as well. This is achieved in one of the 2 ways:

1. One-vs-One classification: It builds a binary classification model for each pair of classes. Thus, there will be n \* (n-1) / 2 models, where n is the number of classes. So, if there are 3 classes (as in our case), then (3\*2)/2=3 models are used.
2. One-vs-All classification: It compares every class with the remaining classes thereby building a model for every class. The class with the highest probability is chosen. So, if there are 3 classes (in our case) then 3 models are used.

The below demo shows the classification of Iris dataset using SVM for multiple classes – setosa, versicolor and virginica.

**Feature Engineering**

You can encode the species column with numerical values. And replace label 'setosa' with '0', 'versicolor' with '1' and 'virginica' with '2'. This encoding technique, of converting distinct values of column into unique numbers is called 'label encoding'.

1. *# encoding the species column*
2. iris\_data.loc[iris\_data.Species=="setosa","Species"] = 0
3. iris\_data.loc[iris\_data.Species=="versicolor","Species"] = 1
4. iris\_data.loc[iris\_data.Species=="virginica","Species"] = 2
5. *#data type (dtype) of the column will be converted to 'category'*
6. iris\_data.Species = iris\_data.Species.astype("category")
7. *#https://pandas.pydata.org/pandas-docs/stable/user\_guide/categorical.html*
8. iris\_data.head()

**Multiclass Model Building using SVM**

1. X = iris\_data[["Petal.Length","Petal.Width"]]
2. Y = iris\_data["Species"]
3. model = SVC()
4. model.fit(X,Y)

**Visualization of Model**

1. from mlxtend.plotting import plot\_decision\_regions
2. features = np.array(X)
3. target = np.array(Y)
4. plot\_decision\_regions(features,target,clf=model)
5. plt.xlabel("Petal length")
6. plt.ylabel("Petal width")
7. plt.title('Multiclass classification on iris using SVM')

A graph of different colored triangles

Description automatically generated with medium confidence

SVM usually performs well where -

* The dataset has fewer number of classes (preferably 2) in the target variable.
* The dataset is high dimensional.
* The dataset is balanced. For example, Iris dataset is a balanced dataset because it has almost equal number of instances in each class/category.

**Code:**[Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600398349197312188/web-hosted/assets/SupportvectorMachines1603472787680.zip) to download the code discussed in Support Vector Machine.

Cross Validation and Ensemble methods

# 1/5 Tuning model hyper parameters using cross-validation

Various machine learning models have been built using algorithms such as Regression, k-Nearest Neighbors, Decision Trees and SVM.

However, while building some of these models, situations such as overfitting or underfitting were encountered. These can be controlled by tuning the hyper parameters of the model.

In this section, we shall look at **cross validation** as one of the approaches to tuning the hyperparameters of the model.

While tuning models, the training data is divided into subsets - train set and **validation set**. The idea is that the model is trained with different hyperparameters on the train set and its performance is measured on the validation set. The validation set acts as a simulation of the test data. However, the actual test data is not used while model building or tuning.

Considering the fact that setting aside a separate validation set might further reduce the data from which the machine can learn, we use a popular approach called cross-validation.

One of the widely used cross-validation techniques is **k-fold cross-validation** where the training data is divided into k equal sized subsamples. Among these subsamples, k-1 subsamples are selected for training and the remaining subsample is used for validation. This procedure is repeated k times such that each subsample is used exactly once as a validation dataset. The results are aggregated to get the final prediction.

## 2/5 ****Implementing cross validation****

The below code demonstrates the usage of 5 fold cross validation to determine the best k value for a kNN model built on the defaulter dataset.

The dataset can be downloaded [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600397281394688191/web-hosted/assets/default1603042568436.zip).

**Reading the data**

The defaulter dataset, contains data about customers defaulting on loans.

1. *#read data from input csv file*
2. defaulter = pd.read\_csv("datasets/defaulter.csv")

**Feature Engineering**

We will now normalize the features in the dataset using MinMaxScaler

1. *#### Normalizing the data using MinMaxScaler*
2. from sklearn.preprocessing import MinMaxScaler
3. scaler = MinMaxScaler()
4. features\_to\_scale = ["balance","income"]
5. scaled\_values = scaler.fit\_transform(defaulter[features\_to\_scale])
6. defaulter["norm\_balance"] = scaled\_values[:,0]
7. defaulter["norm\_income"] = scaled\_values[:,1]

**Splitting the data into train and test set**

1. from sklearn.model\_selection import train\_test\_split
2. X=defaulter[["norm\_balance","norm\_income"]]
3. Y=defaulter['defaulter']
4. X\_train,X\_test,Y\_train,Y\_test = train\_test\_split(X,Y,test\_size=0.2,random\_state=100)

**Finding best value of k for KNN**

1. from sklearn.neighbors import KNeighborsClassifier
2. from sklearn.model\_selection import GridSearchCV
3. *#create new a knn model*
4. knn = KNeighborsClassifier()
5. *#create a dictionary of all k neighbor values*
6. param\_grid = {'n\_neighbors': np.arange(1, 15,2)}
7. '''using GridSearchCV to perform k-fold validation'''
8. knn\_gscv = GridSearchCV(knn, param\_grid,return\_train\_score=True, verbose=1,scoring='accuracy')
9. *#fit model to data*
10. knn\_gscv.fit(X\_train,Y\_train)
11. *#storing results to dataframe*
12. *#print(knn\_gscv.cv\_results\_)*
13. df=pd.DataFrame(knn\_gscv.cv\_results\_)
14. *#filtering out columns*
15. df=df[['param\_n\_neighbors','mean\_train\_score','mean\_test\_score']]

A screenshot of a graph

Description automatically generated

The model parameter 'param\_n\_neighbors' is used to set the value of k of KNN.

Here we observe that, for param\_n\_neighbors = 9, you will get good performance on test and train data. So choose 9 as best value for k.

3/5 Having determined the best value of k using 5 fold cross-validation, use that value to train a model on the entire training data and check the performance on train and test data as shown below:

1. model = KNeighborsClassifier(n\_neighbors = 9, metric="euclidean")
2. model.fit(X\_train,Y\_train)
3. train\_accuracy = model.score(X\_train,Y\_train)
4. test\_accuracy = model.score(X\_test,Y\_test)
5. print(train\_accuracy,test\_accuracy)
6. *#output*
7. *#0.971 0.9685*

**Code:**[Click here to download](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600397281394688191/web-hosted/assets/CrossValidation1604986337228.zip) the code for cross-validation.

# 4/5 Ensemble methods

Ensemble methods are techniques that aim at improving the prediction accuracy in models by creating and combining multiple models instead of using a single model.

Two commonly used ensemble methods are **Bagging** and **Boosting**.

# Bagging

In Bagging, multiple models are trained using the same algorithm on different subsets of the training data. Once multiple models are trained in this manner, they are aggregated using maximum voting or simple aggregation methods such as averaging.

**Random forest** is a special type of bagging algorithm which uses decision trees as base models. It creates random subsets of the training dataset to create a collection of trees. While building a tree, it also randomly samples the feature variables at each split. This adds another layer of variety and randomness to the final classifier. While using the random forest on a new data, the new data is run through each of the trees in the collection and the target predictions from them are aggregated to give the final output.

In the demo code you will see training a random forest model with 10 decision trees.

**Reading the input data**

Here, you will work on spambase dataset, where the normalized frequency of different words in an email are recorded, based on which an email is labelled as spam (1) or not spam (0). [Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600397281394688191/web-hosted/assets/spambase1604748714132.zip) to download the dataset.

1. *#reading input data from csv file*
2. spam\_data = pd.read\_csv("datasets/spambase.csv")

**Splitting the data into train and test set**

1. from sklearn.model\_selection import train\_test\_split
2. features = spam\_data.columns.drop('spam')
3. target = "spam"
4. X=spam\_data[features]
5. Y=spam\_data[target]
6. X\_train,X\_test,Y\_train,Y\_test = train\_test\_split(X,Y,test\_size=0.2,random\_state=100)

**Model Building**

1. from sklearn.ensemble import RandomForestClassifier
2. *# building model with RandomforestClassifier with 10 underlying Decision tree models/ estimators*
3. model = RandomForestClassifier(n\_estimators=10,
4. min\_samples\_split=20,
5. min\_impurity\_decrease=0.05)
6. model.fit(X\_train,Y\_train)
7. *# Evaluate the model performance*
8. train\_accuracy = model.score(X\_train,Y\_train)
9. test\_accuracy = model.score(X\_test,Y\_test)
10. print(train\_accuracy,test\_accuracy)
11. *#output*
12. *#0.8633152173913043 0.8577633007600435*

**Reviewing the feature imporatance**

Random forest model can also help us evaluate which features are important. The below code demonstrates the same.

1. feature\_imps = pd.DataFrame(np.array([features,
2. model.feature\_importances\_]).T,
3. columns=["feature","importance"])
4. feature\_imps.sort\_values(by="importance",ascending=False)

In the above code,  the model.feature\_importances\_ value is used to determine the importance of each feature in the random forest model.

The random forest model found 10 features to be useful out of 50+ features in the dataset.

# 5/5 Boosting

Boosting is another ensemble learning technique where the models are built sequentially. Each new model is built by taking into account the mistakes made by the previous model in predicting target value. This is done by assigning the same weight to each training sample at the beginning. The samples which get incorrectly labelled by a model are given more weight while building the subsequent model. Output of a boosted model is the weighted sum of the predictions made by the individual models. AdaBoost is one of the well known boosting techniques.

The below code shows how you can use **Adaboost** in sklearn.

1. from sklearn.tree import DecisionTreeClassifier
2. from sklearn.ensemble import AdaBoostClassifier
3. *#building AdaBoostClassifier with 10 models, also called as estimators.*
4. model = AdaBoostClassifier(n\_estimators=10)
5. model.fit(X\_train,Y\_train)
6. *# Evaluating the model performance*
7. train\_accuracy = model.score(X\_train,Y\_train)
8. test\_accuracy = model.score(X\_test,Y\_test)
9. print(train\_accuracy,test\_accuracy)
10. *#output*
11. *#0.9195652173913044 0.9272529858849077*

Similiar to random forest, the adaboost classifier also exposes the important features.

1. feature\_imps = pd.DataFrame(np.array([features,
2. model.feature\_importances\_]).T,
3. columns=["feature","importance"])
4. feature\_imps.sort\_values(by="importance",ascending=False)

**Code:** [Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600397281394688191/web-hosted/assets/EnsembleMethods1603483577483.zip) to download the code for ensemble methods

# Exercise

For the Banknote Authentication dataset available in UCI Machine Learning Repository, build models based on Random Forest Classifier and Adaptive Boosting Classifier, to predict whether a banknote will be classified as class 0 (i.e. fake) or class 1 (i.e. genuine) . Compare the performance of the models with the decision  tree model and the logistic regression model built as part of earlier exercises in the course.

[Click here](https://lex.infosysapps.com/apis/authContent/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600400291553280199/web-hosted/assets/databanknoteauthentication1603041947207.zip) to download the data.

Clustering analysis

# 1/2 Clustering

Now that you are familiar with various algorithms that help in supervised learning. Next, Let us briefly discuss unsupervised learning with an example of clustering.

Unsupervised learning deals with historical data which contains no labels. For example the handwritten digits dataset contains pixel intensities of digits captured as 28x28 images. These images do not have any labels associated with them.

The handwritten digits dataset can be downloaded [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600398700904448234/web-hosted/assets/mnist_data.csv).

**Reading the data**

The below code helps you understand the data. Here you can see  the plot of the first data point in the set and it happens to be an image of the digit 2.

1. *# Load the data from the file*
2. mnist\_data = pd.read\_csv("datasets/mnist\_data.csv")
3. *# Plotting the first image in the data*
4. im = np.asarray(mnist\_data.iloc[0:1,:]).reshape(28,28)
5. plt.imshow(im,cmap=plt.cm.gray)

A white and black pixelated image

Description automatically generated

Now let us employ clustering to group similar-looking data points (digits). The aim of clustering is to group similar records together and make sure that the members of different groups are significantly different from each other.

Clustering can be performed using several algorithms and one of the widely used clustering algorithms is the **K-means** algorithm. The K-means algorithm works on numerical data and uses Euclidean distance to identify which data points are close to one another.

**Building K-means clustering model**

In the following code, we are building a model based on the K-means clustering algorithm.

1. from sklearn.cluster import KMeans
2. *# building K-Means model with 10 clusters*
3. model = KMeans(n\_clusters=10)
4. model.fit(mnist\_data)
5. *# Take a look at the cluster labels that are generated*
6. *# Note that these cluster labels do not indicate the digit in the images*
7. print(np.unique(model.labels\_))
8. *#output*
9. *#[0 1 2 3 4 5 6 7 8 9]*

2/2 **Exploring the clusters generated by the model:**

Let us now look at a few data points grouped by the model.

1. *# Explore images in cluster 1*
2. *# cluster1 variable holds the data that has been grouped into the first cluster*
3. cluster1 = mnist\_data[model.labels\_==0]
4. *# Pick 5 random images from cluster 1*
5. cluster1\_imgs = cluster1.iloc[[np.random.randint(0,cluster1.shape[0]) for i in range(0,5)]]
6. *# Plot the images in cluster 1*
7. for i in range(0,cluster1\_imgs.shape[0]):
8. plt.subplot(1,5,i+1)
9. img\_fig = np.asarray(cluster1\_imgs[i:i+1]).reshape(28,28)
10. plt.imshow(img\_fig,cmap=plt.cm.gray)

A number in black squares

Description automatically generated

1. *# Explore images in cluster 2*
2. cluster2 = mnist\_data[model.labels\_==1]
3. cluster2\_imgs = cluster2.iloc[[np.random.randint(0,cluster2.shape[0]) for i in range(0,5)]]
4. for i in range(0,cluster2\_imgs.shape[0]):
5. plt.subplot(1,5,i+1)
6. img\_fig = np.asarray(cluster2\_imgs[i:i+1]).reshape(28,28)
7. plt.imshow(img\_fig,cmap=plt.cm.gray)

A black and white image of a number

Description automatically generated

Applications of clustering

# Clustering based on Topics

Documents such as white papers, research papers, reports etc. are published in great quantities and there is a need to cluster and visualize them based on topics.

# Information Retrieval

Clustering can be used to group websites or documents based on the contents. Suppose if there is a hit on a website or document then other websites or documents in that cluster are also likely to be relevant.

# Anomaly Detection

The aim of anomaly detection is to find objects that are significantly different from the rest. This is useful in detecting fraudulent transactions in finance and banking, intrusion detection, etc.

**Code:**[Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012600398700904448234/web-hosted/assets/Clustering1603474335425.zip) to download the code discussed in Clustering.

Introduction to Artificial Neural Network

1/2 Neural Networks

Even with the advancements of technology and algorithms, there is still no match to the capabilities of the human brain which hasn't been fully understood yet. The core components of our nervous system or the brain are the brain cells or nerve cells which are also referred as “neurons”. These cells are connected to one another to form a complex network structure known as “**Neural Network**”.

A diagram of a cell body

Description automatically generated

The above figure represents the structure of a neuron which has three major components, namely,

1. Cell body
2. Dendrite
3. Axon

Dendrites receive signal from other neurons and bring it to the cell body, where the signal is processed. The axon transmits the processed signal from the cell body to dendrites of the neighboring cells to which it is connected.

2/2  The study of **artificial neural network** (ANN) is inspired by attempts to simulate the biological neural system. ANN consists of interconnected artificial neurons or **nodes**, analogous to human/biological neuron network. Just like a biological neuron receives an input signal, processes it and transmits the output to other neurons, in ANN, a node receives the input, processes it using a function known as **activation function** and transmits the output to other nodes.

The figure below shows the analogy between a neuron of the human brain and the node of ANN.

A diagram of a machine

Description automatically generated

The figure below shows part of a biological neural network and the Artificial neural network (ANN).

A diagram of a network of neurons

Description automatically generated

Details of ANN

# 1/2 **Components of ANN**

**Artificial Neurons/Nodes:** These are the elementary units in a neural network. An artificial neuron is a computational unit.

**Layers:** These are group of neurons at different levels. An ANN has one input layer, one or more hidden layers and one output layer. Nodes in input layer are the nodes to which we feed in the input feature values. Output layer nodes give us the output or target values. The layers between input and output layers are called as hidden layers.

**Weights and Biases:** Weights are numerical parameters which determine how strongly each of the neurons affects the other. Bias is special neuron with the value 1. It is added to each pre-output layer.

**Activation function:** An activation function is a non-linear mathematical function which converts the input values to an output. Without activation functions, the working of a neural network will be like linear models.

2/2 Lets now consider an example to demonstrate how an ANN is used.

Assume we have a lot of past medical data about diabetic patients. We want to make a model to predict whether a person is “Diabetic” or “Non-Diabetic” based on the inputs, “Sugar Level” and “Age”.

We will see how an ANN is trained with the past data for this purpose.

**Black Box Example -  Diabetic prediction:**

The example we considered can be compared to black box to predict whether the person is “Diabetic” or “Non- Diabetic” on the following input.

**INPUT:**

Sugar level

Age

**OUTPUT:**

Diabetic

Non- Diabetic

A black cube with white text and blue squares

Description automatically generated

## ****Layers in ANN****

The problem can be modelled as a neural network as shown below. Let us learn about each of the layers in detail to understand how the model works. Note that the input to the model is the age and the sugar level and the output is a binary decision if a patient is diabetic

A diagram of a network

Description automatically generated

**Input Layer**

* It is the first layer in any neural network.
* It brings the initial data into the system for further processing by subsequent layers of artificial neurons. (It is the only layer where no computation happens i.e. no activation function is applied.)
* Here we will pass “Sugar Level” and “Age” as input parameters to the neurons in the input layer. A diagram of a structure

  Description automatically generated

**Hidden Layer**

* Layers between input layer and output layer in any ANN are called hidden layers.
* It takes a set of weighted input and applies a non-linear activation function to it.
* Each of these hidden layers can contain same or different number of neurons.

A diagram of a layer

Description automatically generated

**Output Layer**

* It is the last layer in any neural network, from which we get the final target values.
* Usually, output layer nodes also have activation fuctions.
* The neurons of this layer will give us the output whether the person is “Diabetic” or “Non-Diabetic”.

A diagram of a diagram

Description automatically generated

## 2/2 ****Learning Weights and Biases****

Weights and biases are parameters that are learned (or adjusted) to produce the desired outputs from the given inputs. Mathematical techniques like stochastic Gradient Descent, Adam etc. are used to search for the optimal set of weights that will make the model accurate in its prediction.

A diagram of a network

Description automatically generated

## ****Activation Function****

An activation function (also known as **transfer function**) is a non-linear mathematical function which converts input values to an output. It helps neural networks find non-linear relations in data. An activation can also be a linear function. When a linear activation function is used, then such a neural network function as linear regression. Commonly used functions:

* Linear activation function
* Binarystep
* Sigmoid
* Tanh
* ReLU
* Softmax
* ​​​​​​​Softplus or smoothRelu

A group of graphs with red lines

Description automatically generated

A graph of a function

Description automatically generated

**Activation function**

**Formula**

**Linear**



**Binary step**



​​​​​​

**Sigmoid**



**Tanh**



**Relu**



**Softmax**



**Softplus or Smooth Relu**



**LeakyRelu**

​​​​​​​

A table with math equations

Description automatically generated

## ****The process of Learning****

The model learns its parameters ( weights and biases ) through a process of forward propagation of data and back propagation of errors.

To start with, random weights and biases are assumed. The input is then fed forward through the network. Each hidden layer modifies the input forwarded to it till it reaches the output. This output  achieved is compared against the expected output. Error is recorded as their difference. This error is then propagated backward through the layers to adjust the parameters i.e. the weights and the biases. The forward pass and the back propagation continues over a number of iterations till a desirable low level of error is achieved. The neural network is then said to have learnt its parameters.

A diagram of a diagram

Description automatically generated

Building an Artificial Neural Network

1/3 Consider the MNIST handwritten dataset. Let us now look at how a Neural network can be used to classify this data.

The MNIST dataset can be downloaded [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_0131395375426764803690/web-hosted/assets/datasets1603947951084.zip).

The below code demonstrates the usage of MLPClassifier in sklearn.neural\_network that helps us create a classifier using a neural network.

Reading the data

1. *# Train data contains digit data and the correct labels*
2. *# Test data contains just the digit data and no labels*
3. mnist\_train = pd.read\_csv("datasets/mnist/train.csv")
4. mnist\_test = pd.read\_csv("datasets/mnist/test.csv")
5. *# Let's visualize the image represented by the first rows of the train data and the test data*
6. train\_data\_digit1 = np.asarray(mnist\_train.iloc[0:1,1:]).reshape(28,28)
7. test\_data\_digit1 = np.asarray(mnist\_test.iloc[0:1,]).reshape(28,28)
8. plt.subplot(1,2,1)
9. plt.imshow(train\_data\_digit1,cmap = plt.cm.gray\_r)
10. plt.title("First digit in train data")
11. plt.subplot(1,2,2)
12. plt.imshow(test\_data\_digit1,cmap = plt.cm.gray\_r)
13. plt.title("First digit in test data ")

A black and white image of a number and a number

Description automatically generated with medium confidence

### 2/3 Feature Engineering

1. """Let us now assign the label column value to a new variable Y\_train
2. and the remaining column values to X\_train"""
3. X\_train = mnist\_train.iloc[:,1:]
4. Y\_train = mnist\_train.iloc[:,0:1]

### Building an Artificial Neural Network

1. from sklearn.neural\_network import MLPClassifier
2. *# Let us now create a neural network model to learn from train data*
3. *# We shall build a single hidden layer with 50 nodes.*
4. nn\_model = MLPClassifier(hidden\_layer\_sizes=(50))
5. *# The fit method initiates the learning process. When its execution completes, the model is learnt*
6. nn\_model.fit(X\_train,mnist\_train.iloc[:,0])
7. *# Now that we have a model, lets get it to predict the value of the first digit in the test data*
8. print(nn\_model.predict(mnist\_test.iloc[0:1,]))
9. *# output*
10. *# [2]*

### 3/3 Evaluate model performance

1. from sklearn.metrics import classification\_report
2. *# report based on tain data*
3. print(classificati

A screenshot of a computer screen

Description automatically generated

The code and the dataset used in the demo can be downloaded [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_0131395375426764803690/web-hosted/assets/Neuralnetworks1603947845970.zip).

Summary

# In this course we have learned:

* What is machine learning?
* Supervised and unsupervised learning
* Regression and its application
* Classification and its applications
* Clustering and its applications
* Overview of artificial neural networks

# References used in making this course

* Applied logistic regression:  David W. Hosmer (University of Massachussets Amhrest), Stanley Lemeshow (The Ohio State University Columbus).
* Data Analysis Using Regression and Multilevel/Hierarchical Models: Andrew Gelman, Jennifer Hill
* Regression Analysis by Example: SAMPRIT CHATTERJEE (Department of Health Policy Mount Sinai School of Medicine, New York, NY), ALI S . HADI Department of Mathematics The American University in Cairo, Cairo, Egypt
* Applied Regression Analysis: Norman R. Draper, Harry Smith.
* Data Mining Concepts and Techniques, 3rd Edition by, J. Han, M Kamber, J Pei

## Capstone Project: Explore Machine Learning using Python

When an employee quits the organization, they take way experience, skill, knowledge acquired over a period of time within the organization. This affects the organization and the impact is not only restricted to that but also brings the task of finding a suitable replacement. Mostly the suitable replacement is hired from external and it again adds time and cost to the organization.

The HR department of a multinational company would like to understand the reasons for premature exit of experienced employees using Machine Learning techniques. For achieving this, they must:

* Explore the dataset and check if the data can be used as-is.
* Determine the relationship between satisfaction level and working hours of employees who have left the organization.
* Understand the effect of satisfaction level, department, promotion in last 5 years and salary level of employees who have left the organization.
* Build  a machine learning model to predict  the exit of employees.

[Click here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_0131364956063416323344/web-hosted/assets/HRcommasep1603576336980.zip) to download the dataset:

The dataset has roughly 15000 records with10 columns, which are self-explanatory, namely: satisfaction\_level, last\_evaluation, number\_project, average\_monthly\_hours, time\_spend\_company, Work\_accident, left, promotion\_last\_5years, Department, salary.