Unit 5 Issues in Decision Tree

OUTLINE:

- Issues in Decision Tree

Issues in Decision Tree

- Determining how deeply to grow the decision tree
- Handling continuous attributes
- Choosing an appropriate attribute selection measure
- Handling training data with missing attribute values
- Handling attributes with differing costs

Decision Tree Construction

What can we do to avoid overfitting? Here are a few examples of frequent heuristics:

- Don't try to fit all of the examples in; instead, quit before the training set runs out.
- After fitting all of the instances, prune the resulting tree.
- In decision tree learning, there are numerous methods for **preventing overfitting.**
- These may be divided into two categories:
- Techniques that stop growing the tree before it reaches the point where it properly classifies the training data.
- Then post-prune the tree, and ways that allow the tree to overfit the data and then **post-prune** the tree.
- Despite the fact that the first strategy appears to be more straightforward, the second approach of post-pruning overfit trees has shown to be more effective in reality.

Decision Tree Construction

- **The criterion used to determine the correct final tree size:**
- To assess the usefulness of post-pruning nodes from the tree, use a separate set of examples from the training examples.
- Use all available data for training, but do a statistical test to see if extending (or pruning) a specific node would result in a better result than the training set.
- A chi-square test is performed to see if enlarging a node would increase performance throughout the full instance distribution or only on the current sample of training data.
- When encoding the training samples and the decision tree, use an explicit measure of complexity, with the tree's development halted when the encoding size is reduced. This method is based on the Minimum Description Length concept, which is a heuristic.

Unit 5 Decision Tree Algorithm

OUTLINE:

- ❖ Decision Tree Algorithm

Decision Tree Algorithm

How does the Decision Tree algorithm Work?

In a decision tree, for predicting the class of the given dataset, the algorithm starts from the root node of the tree.

This algorithm compares the values of root attribute with the record (real dataset) attribute and, based on the comparison, follows the branch and jumps to the next node.

For the next node, the algorithm again compares the attribute value with the other sub-nodes and move further.

It continues the process until it reaches the leaf node of the tree.

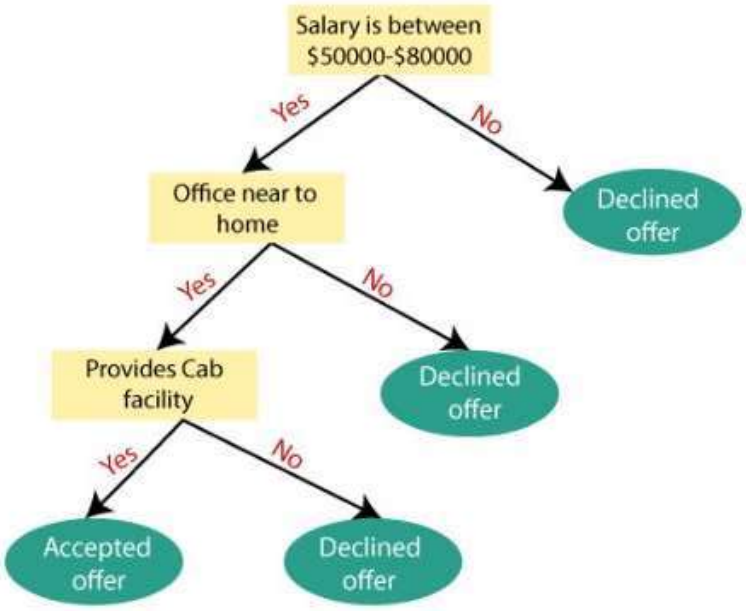
The complete process can be better understood using the below algorithm:

Decision Tree Algorithm

Step-1: Begin the tree with the root node, says S, which contains the complete dataset.
Step-2: Find the best attribute in the dataset using **Attribute Selection Measure (ASM).**
Step-3: Divide the S into subsets that contains possible values for the best attributes.
Step-4: Generate the decision tree node, which contains the best attribute.
Step-5: Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

Decision Tree Algorithm

Example: Suppose there is a candidate who has a job offer and wants to decide whether he should accept the offer or Not. So, to solve this problem, the decision tree starts with the root node (Salary attribute by ASM). The root node splits further into the next decision node (distance from the office) and one leaf node based on the corresponding labels. The next decision node further gets split into one decision node (Cab facility) and one leaf node. Finally, the decision node splits into two leaf nodes (Accepted offers and Declined offer). Consider the below diagram:



Decision Tree Algorithm

Attribute Selection Measures:

While implementing a Decision tree, the main issue arises that **how to select the best attribute for the root node and for sub-nodes**. So, to solve such problems there is a technique which is called as **Attribute selection measure or ASM.** By this measurement, we can easily select the best attribute for the nodes of the tree. There are two popular techniques for ASM, which are:

1. **Information Gain**
2. **Gini Index**

Decision Tree Algorithm

Information Gain:

- Information gain is the measurement of changes in entropy after the segmentation of a dataset based on an attribute.
- It calculates how much information a feature provides us about a class.
- According to the value of information gain, we split the node and build the decision tree.
- A decision tree algorithm always tries to maximize the value of information gain, and a node/attribute having the highest information gain is split first. It can be calculated using the below formula:

Information Gain= Entropy(S)- [(Weighted Avg) *Entropy(each feature)]

Entropy: Entropy is a metric to measure the impurity in a given attribute. It specifies randomness in data. Entropy can be calculated as:

Entropy(s)= -P(yes)log2 P(yes)- P(no) log2 P(no)**Where,**

S= Total number of samples

P(yes)= probability of yes

P(no)= probability of no

Decision Tree Algorithm

Gini Index:

- Gini index is a measure of impurity or purity used while creating a decision tree in the CART(Classification and Regression Tree) algorithm.
- An attribute with the low Gini index should be preferred as compared to the high Gini index.
- It only creates binary splits, and the CART algorithm uses the Gini index to create binary splits.
- Gini index can be calculated using the below formula:

$$\text{Gini Index} = 1 - \sum_j P_j^2$$

Decision Tree Algorithm

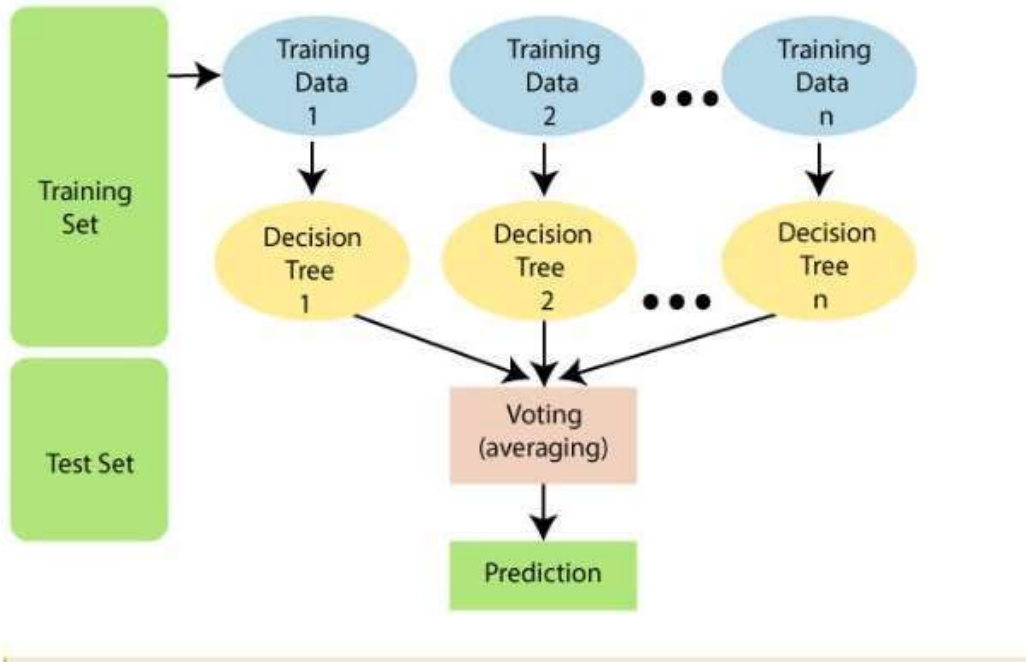
- **Pruning: Getting an Optimal Decision tree**
- *Pruning is a process of* **deleting the unnecessary nodes from a tree in order to get the optimal decision tree.**
- A too-large tree increases the risk of overfitting, and a small tree may not capture all the important features of the dataset.
- Therefore, a technique that decreases the size of the learning tree without reducing accuracy is known as Pruning.
- There are mainly two types of tree **pruning** technology used:

1. **Cost Complexity Pruning**
2. **Reduced Error Pruning.**

Random Forest Algorithm

- Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML.
- It is based on the concept of **ensemble learning,** which is a process of *combining multiple classifiers to solve a complex problem and to improve the performance of the model.*
- As the name suggests, ***"Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset."*** Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.
- **The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.**

Random Forest Algorithm



Random Forest Algorithm

Assumptions for Random Forest

Since the random forest combines multiple trees to predict the class of the dataset, **it is possible that some decision trees may predict the correct output,** while **others may not**. But together, **all the trees predict the correct output**. Therefore, below are two assumptions for a better Random forest classifier:

- There should be some actual values in the feature variable of the dataset so that the classifier can predict accurate results rather than a guessed result.
- The predictions from each tree must have very low correlations.

Random Forest Algorithm

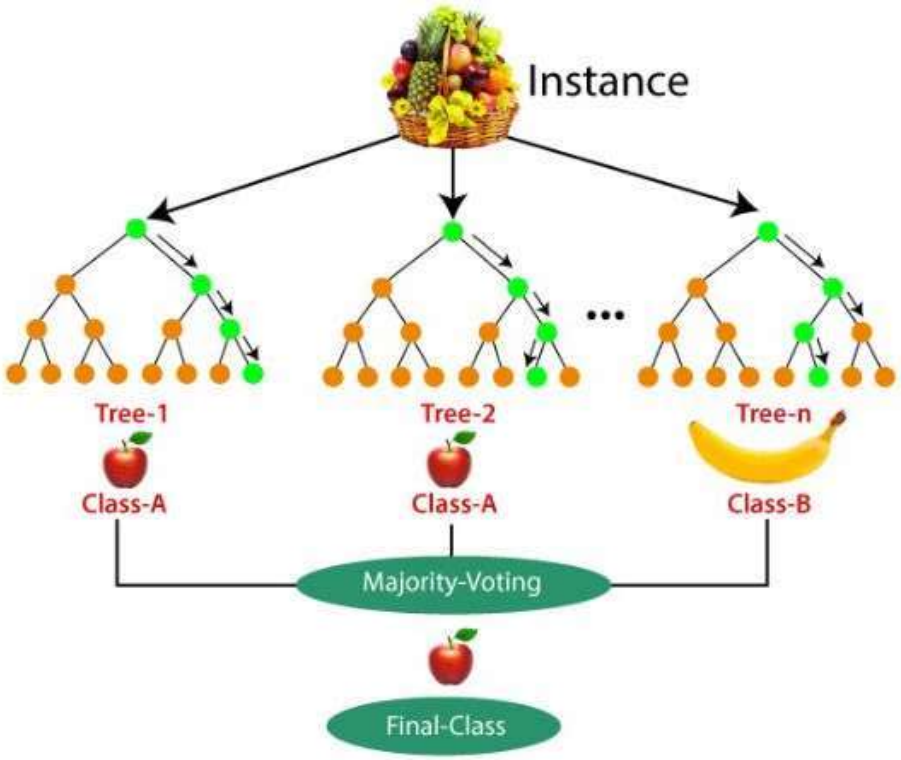
Why use Random Forest?
Below are some points that explain why we should use the Random Forest algorithm:
<=”” li=””>It takes **less training time** as compared to other algorithms.
It predicts **output with high accuracy**, even for the large dataset it runs efficiently.
It can also maintain accuracy **when a large proportion of data is missing.**

Random Forest Algorithm

How does Random Forest algorithm work?
Random Forest works in two-phase first is to create the random forest by combining N decision tree, and second is to make predictions for each tree created in the first phase.
The Working process can be explained in the below steps and diagram:
Step-1: Select random **K data points from the training set.**
Step-2: Build the **decision trees associated with the selected data** points (Subsets).
Step-3: Choose **the number N for decision trees that you want to build**.
Step-4: Repeat Step 1 & 2.
Step-5: For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes.

Random Forest Algorithm

Example: Suppose there is a **dataset that contains multiple fruit images**. So, this **dataset is given to the Random forest classifier**. The dataset is divided into subsets and given to each decision tree. **During the training phase**, each decision tree **produces a prediction result**, and **when a new data point occurs**, then based on the **majority of results, the Random Forest classifier predicts the final decision**. Consider the below image:



Random Forest Algorithm

Python Implementation of Random Forest Algorithm:

- Data Pre-processing step
- Fitting the Random forest algorithm to the Training set
- Predicting the test result
- Test accuracy of the result (Creation of Confusion matrix)
- Visualizing the test set result.

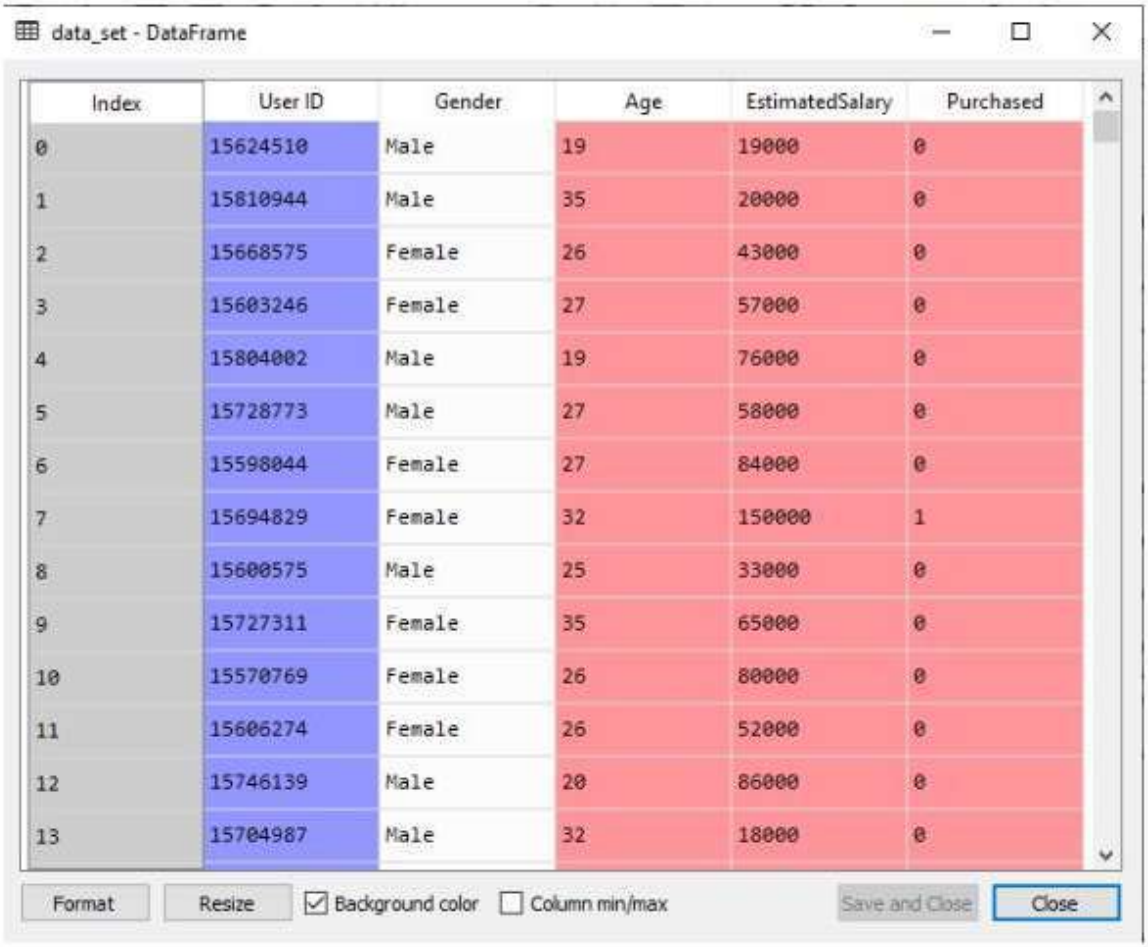
Random Forest Algorithm

1.Data Pre-Processing Step:

Below is the code for the pre-processing step:

```
# importing libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd

#importing datasets
data_set= pd.read_csv('user_data.csv')

#Extracting Independent and dependent Variable
x= data_set.iloc[:, [2,3]].values
y= data_set.iloc[:, 4].values

# Splitting the dataset into training and test set.
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.25, random_state=0)

#feature Scaling
from sklearn.preprocessing import StandardScaler
st_x= StandardScaler()
x_train= st_x.fit_transform(x_train)
```

Random Forest Algorithm



Index	User ID	Gender	Age	EstimatedSalary	Purchased
0	15624510	Male	19	19000	0
1	15810944	Male	35	20000	0
2	15668575	Female	26	43000	0
3	15603246	Female	27	57000	0
4	15804002	Male	19	76000	0
5	15728773	Male	27	58000	0
6	15598044	Female	27	84000	0
7	15694829	Female	32	150000	1
8	15600575	Male	25	33000	0
9	15727311	Female	35	65000	0
10	15570769	Female	26	80000	0
11	15606274	Female	26	52000	0
12	15746139	Male	20	86000	0
13	15704987	Male	32	18000	0

Random Forest Algorithm

2. Fitting the Random Forest algorithm to the training set:

Now we will fit the Random forest algorithm to the training set. To fit it, we will import the **RandomForestClassifier** class from the **sklearn.ensemble** library. The code is given below:

```
#Fitting Decision Tree classifier to the training set
from sklearn.ensemble import RandomForestClassifier
classifier= RandomForestClassifier(n_estimators= 10, criterion="entropy")
classifier.fit(x_train, y_train)
```

In the above code, the classifier object takes below parameters:

- **n_estimators**= The required number of trees in the Random Forest. The default value is 10. We can choose any number but need to take care of the overfitting issue.
- **criterion**= It is a function to analyze the accuracy of the split. Here we have taken "entropy" for the information gain.

Random Forest Algorithm

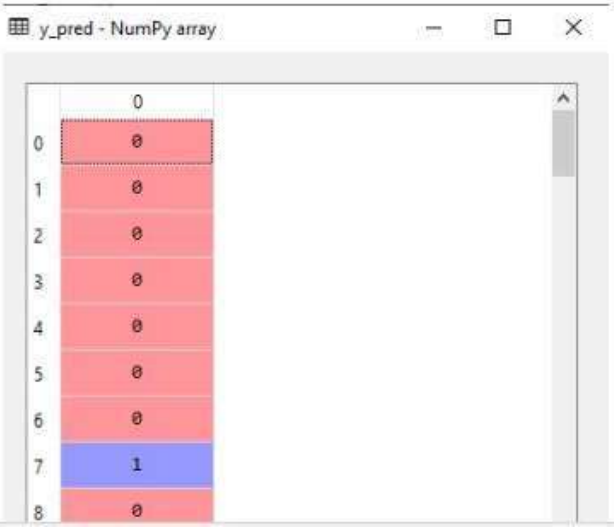
3. Predicting the Test Set result

Since our model is fitted to the training set, so now we can predict the test result. For prediction, we will create a new prediction vector y_pred. Below is the code for it:

```
#Predicting the test set result
y_pred= classifier.predict(x_test)
```

Output:

The prediction vector is given as:



Random Forest Algorithm

4. Creating the Confusion Matrix

Now we will create the confusion matrix to determine the correct and incorrect predictions. Below is the code for it:

```
#Creating the Confusion matrix
from sklearn.metrics import confusion_matrix
cm= confusion_matrix(y_test, y_pred)
```

Random Forest Algorithm

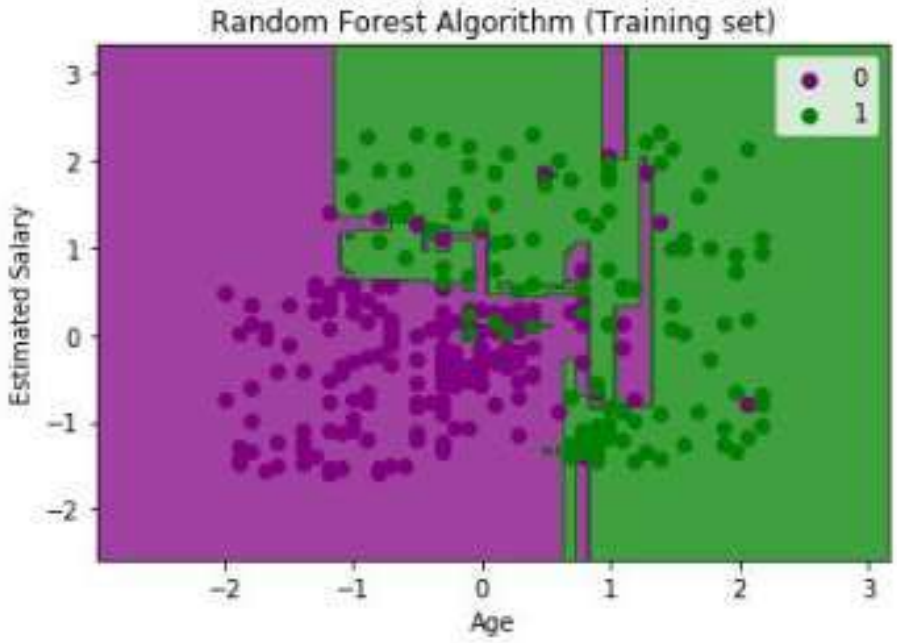
5. Visualizing the training Set result

Here we will visualize the training set result. To visualize the training set result we will plot a graph for the Random forest classifier. The classifier will predict yes or No for the users who have either Purchased or Not purchased the SUV car as we did in Logistic Regression. Below is the code for it:

```
from matplotlib.colors import ListedColormap
x_set, y_set = x_train, y_train
x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max() + 1, step  =0.01),
nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),
alpha = 0.75, cmap = ListedColormap(('purple','green' )))
mtp.xlim(x1.min(), x1.max())
mtp.ylim(x2.min(), x2.max())
for i, j in enumerate(nm.unique(y_set)):
    mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
        c = ListedColormap(('purple', 'green'))(i), label = j)
mtp.title('Random Forest Algorithm (Training set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
mtp.show()
```

Random Forest Algorithm

Output:



Unit 5 Random Forest Classifier using Python Scikit-Learn

OUTLINE:

- **Random Forest Classifier using Python Scikit-Learn**

Random Forest Classifier using Python Scikit-Learn

Steps to Create Random Forest Classifier

We can follow the below steps to create a random forest classifier using Python Scikit-learn −

Step 1 − Import the required libraries.

Step 2 − Load the dataset.

Step 3 − Divide dataset into training and test datasets.

Step 4 − Import random forest classifier from sklearn.ensemble module.

Step 5 − Create dataframe of dataset.

Step 6 − Create a random forest classifier and train the model using fit() function.

Step 7 − Predict from test dataset.

Step 8 − Import metrics to find the accuracy of the classifier.

Step 9 − Print the accuracy of the random forest classifier.

Random Forest Classifier using Python Scikit-Learn

```
# Import required libraries
 import sklearn
 import pandas as pd
from sklearn import datasets

# Load the iris dataset
from sklearn iris_clf = datasets.load_iris()
print(iris_clf.target_names)
print(iris_clf.feature_names)

# Dividing the datasets into training datasets and test datasets
X, y = datasets.load_iris( return_X_y = True)
from sklearn.model_selection import train_test_split
# 60 % training dataset and 40 % test datasets X_train, X_test, y_train,
y_test = train_test_split(X, y, test_size = 0.40)
```

Random Forest Classifier using Python Scikit-Learn

```
# Import random forest classifier
from sklearn assemble module
from sklearn.ensemble
import RandomForestClassifier
# Create dataframe
data = pd.DataFrame({'sepallength': iris_clf.data[:, 0],
'sepalwidth': iris_clf.data[:, 1], 'petallength': iris_clf.data[:, 2], 'petalwidth':
iris_clf.data[:, 3], 'species': iris_clf.target})
# Create a Random Forest classifier
RForest_clf = RandomForestClassifier(n_estimators = 100)
# Train the model on the training dataset by using fit() function
RForest_clf.fit(X_train, y_train)
# Predict from the test dataset
y_pred = RForest_clf.predict(X_test)
# Import metrics for accuracy calculation
from sklearn import metrics
print('\n'"Accuracy of our Random Forst Classifier is: ",
metrics.accuracy_score(y_test, y_pred)*100)
```

Random Forest Classifier using Python Scikit-Learn

Output

It will produce the following output −

```
['setosa' 'versicolor' 'virginica']
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (c

Accuracy of our Random Forst Classifier is: 95.0
```

Unit 5 Classification and Regression Tree (CART)

OUTLINE:

- Classification and Regression Tree (CART)

Classification and Regression Tree (CART)

CART(Classification And Regression Tree) is a variation of the decision tree algorithm. It can handle both classification and regression tasks. Scikit-Learn uses the Classification And Regression Tree (CART) algorithm to train Decision Trees (also called “growing” trees).

A *Classification And Regression Tree* (CART), is a predictive model, which explains how an outcome variable's values can be predicted based on other values. A CART output is a decision tree where each fork is a split in a predictor variable and each end node contains a prediction for the outcome variable.

Classification and Regression Tree (CART)

CART Algorithm

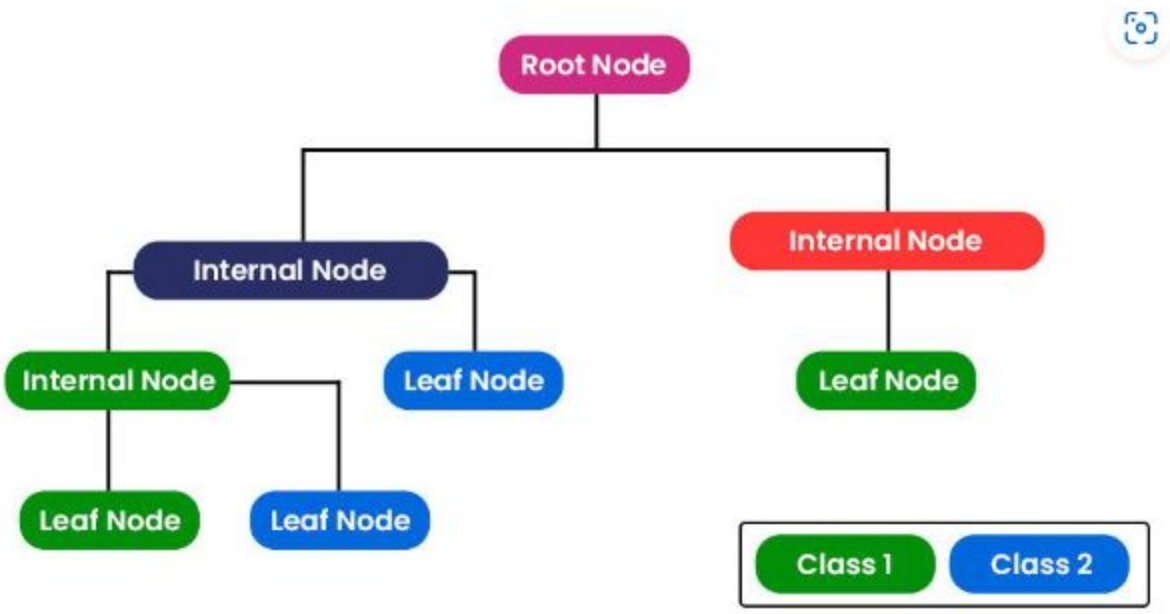
CART is a predictive algorithm used in Machine learning and it explains how the target variable's values can be predicted based on other matters. It is a decision tree where each fork is split into a predictor variable and each node has a prediction for the target variable at the end.

In the decision tree, nodes are split into sub-nodes on the basis of a threshold value of an attribute. The root node is taken as the training set and is split into two by considering the best attribute and threshold value. Further, the subsets are also split using the same logic. This continues till the last pure sub-set is found in the tree or the maximum number of leaves possible in that growing tree.

Classification and Regression Tree (CART)

The CART algorithm works via the following process:

- The best split point of each input is obtained.
- Based on the best split points of each input in Step 1, the new “best” split point is identified.
- Split the chosen input according to the “best” split point.
- Continue splitting until a stopping rule is satisfied or no further desirable splitting is available.



Classification and Regression Tree (CART)

Gini index/Gini impurity

The Gini index is a metric for the classification tasks in CART. It stores the sum of squared probabilities of each class. It computes the degree of probability of a specific variable that is wrongly being classified when chosen randomly and a variation of the Gini coefficient. It works on categorical variables, provides outcomes either “successful” or “failure” and hence conducts binary splitting only.
The degree of the Gini index varies from 0 to 1,
Where 0 depicts that all the elements are allied to a certain class, or only one class exists there.
The Gini index of value 1 signifies that all the elements are randomly distributed across various classes, and
A value of 0.5 denotes the elements are uniformly distributed into some classes.
Mathematically, we can write Gini Impurity as follows:

Classification and Regression Tree (CART)

Classification tree

A classification tree is an algorithm where the target variable is categorical. The algorithm is then used to identify the “Class” within which the target variable is most likely to fall. Classification trees are used when the dataset needs to be split into classes that belong to the response variable(like yes or no)

Regression tree

A Regression tree is an algorithm where the target variable is continuous and the tree is used to predict its value. Regression trees are used when the response variable is continuous. For example, if the response variable is the temperature of the day.

Classification and Regression Tree (CART)

CART model representation

CART models are formed by picking input variables and evaluating split points on those variables until an appropriate tree is produced.

Steps to create a Decision Tree using the CART algorithm:

Greedy algorithm: In this The input space is divided using the Greedy method which is known as a recursive binary spitting. This is a numerical method within which all of the values are aligned and several other split points are tried and assessed using a cost function.

Stopping Criterion: As it works its way down the tree with the training data, the recursive binary splitting method described above must know when to stop splitting. The most frequent halting method is to utilize a minimum amount of training data allocated to every leaf node. If the count is smaller than the specified threshold, the split is rejected and also the node is considered the last leaf node.

Classification and Regression Tree (CART)

CART model representation

Tree pruning: Decision tree's complexity is defined as the number of splits in the tree. Trees with fewer branches are recommended as they are simple to grasp and less prone to cluster the data. Working through each leaf node in the tree and evaluating the effect of deleting it using a hold-out test set is the quickest and simplest pruning approach.

***Data preparation for the CART*:** No special data preparation is required for the CART algorithm.

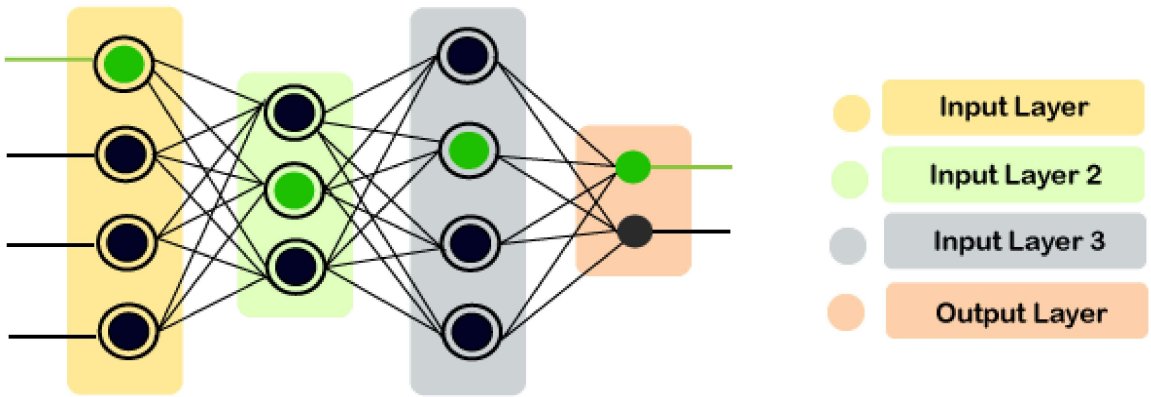
Artificial Neural Networks

At earlier times, the conventional computers incorporated algorithmic approach that is the computer used to follow a set of instructions to solve a problem unless those specific steps need that the computer need to follow are known the computer cannot solve a problem. So, obviously, a person is needed in order to solve the problems or someone who can provide instructions to the computer so as to how to solve that particular problem. It actually restricted the problem-solving capacity of conventional computers to problems that we already understand and know how to solve.

But what about those problems whose answers are not clear, so that is where our traditional approach face failure and so Neural Networks came into existence. Neural Networks processes information in a similar way the human brain does, and these networks actually learn from examples, you cannot program them to perform a specific task. They will learn only from past experiences as well as examples, which is why you don't need to provide all the information regarding any specific task. So, that was the main reason why neural networks came into existence.

> “ Artificial Neural Network is biologically inspired by the neural network, which constitutes after the human brain. ”

Neural networks are modeled in accordance with the human brain so as to imitate their functionality. The human brain can be defined as a neural network that is made up of several neurons, so is the Artificial Neural Network is made of numerous perceptron.

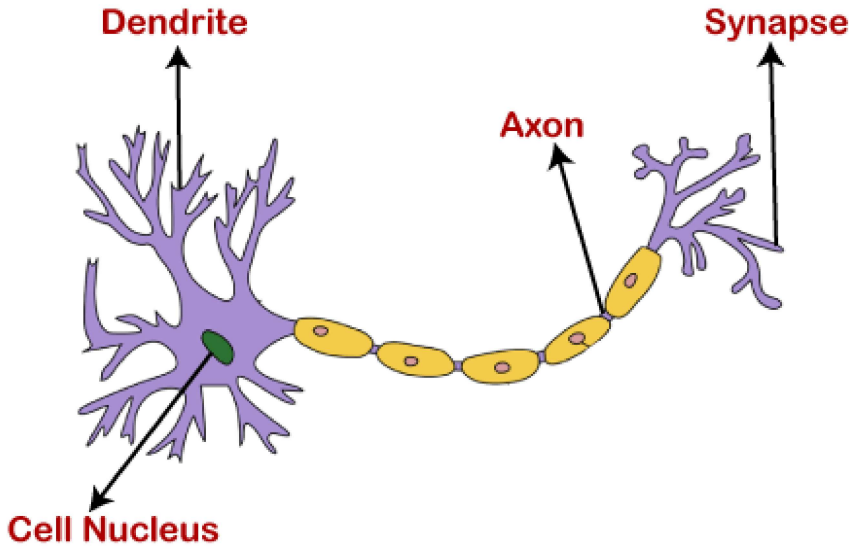


A neural network comprises of three main layers, which are as follows;

- **Input layer:** The input layer accepts all the inputs that are provided by the programmer.
- **Hidden layer:** In between the input and output layer, there is a set of hidden layers on which computations are performed that further results in the output.
- **Output layer:** After the input layer undergoes a series of transformations while passing through the hidden layer, it results in output that is delivered by the output layer.

Motivation behind Neural Network

Basically, the neural network is based on the neurons, which are nothing but the brain cells. A biological neuron receives input from other sources, combines them in some way, followed by performing a nonlinear operation on the result, and the output is the final result.



The **dendrites** will act as a receiver that receives signals from other neurons, which are then passed on to the **cell body**. The cell body will perform some operations that can be a summation, multiplication, etc. After the operations are performed on the set of input, then they are transferred to the next neuron via **axion**, which is the transmitter of the signal for the neuron.

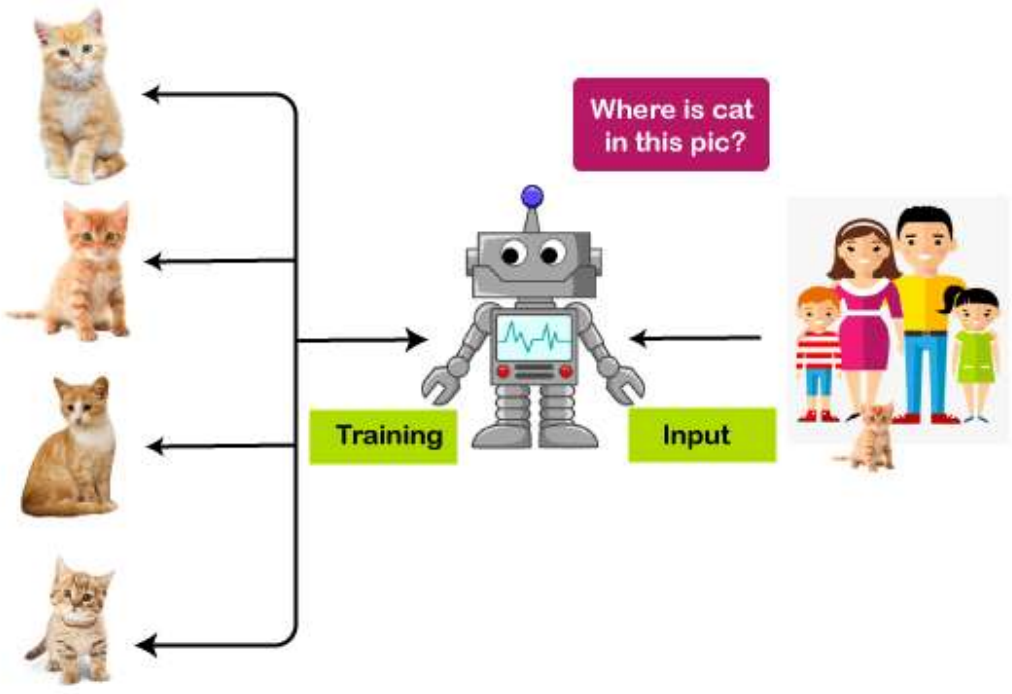
What are Artificial Neural Networks?

Artificial Neural Networks are the computing system that is designed to simulate the way the human brain analyzes and processes the information. Artificial Neural Networks have self-learning capabilities that enable it to produce a better result as more data become available. So, if the network is trained on more data, it will be more accurate because these neural networks learn from the examples. The neural network can be configured for specific applications like data classification, pattern recognition, etc.

With the help of the neural network, we can actually see that a lot of technology has been evolved from translating webpages to other languages to having a virtual assistant to order groceries online. All of these things are possible because of neural networks. So, an artificial neural network is nothing but a network of various artificial neurons.

Importance of Neural Network:

- **Without Neural Network:** Let's have a look at the example given below. Here we have a machine, such that we have trained it with four types of cats, as you can see in the image below. And once we are done with the training, we will provide a random image to that particular machine that has a cat. Since this cat is not similar to the cats through which we have trained our system, so without the neural network, our machine would not identify the cat in the picture. Basically, the machine will get confused in figuring out where the cat is.



- **With Neural Network:** However, when we talk about the case with a neural network, even if we have not trained our machine with that particular cat. But still, it can identify certain features of a cat that we have trained on, and it can match those features with the cat that is

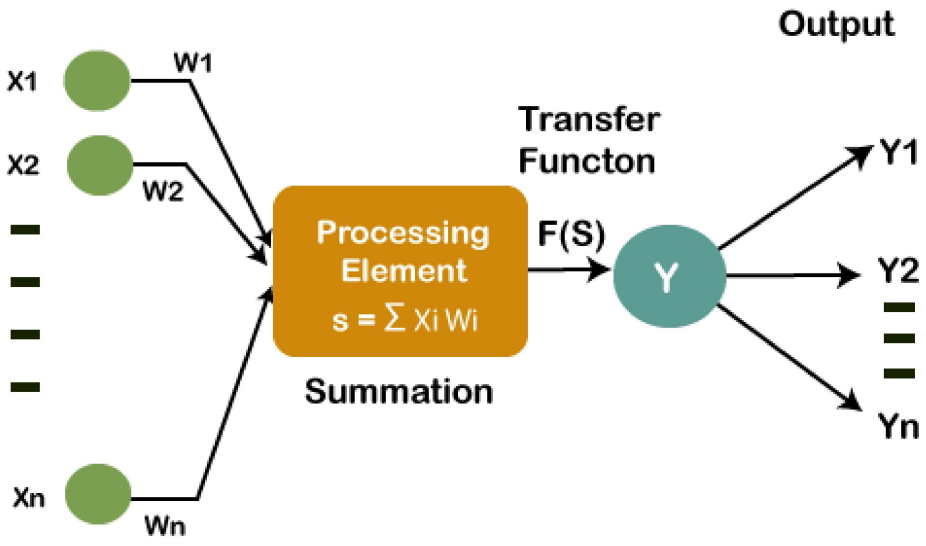
there in that particular image and can also identify the cat. So, with the help of this example, you can clearly see the importance of the concept of a neural network.

Working of Artificial Neural Networks

Instead of directly getting into the working of Artificial Neural Networks, lets breakdown and try to understand Neural Network's basic unit, which is called a **Perceptron**.

So, a perceptron can be defined as a neural network with a single layer that classifies the linear data. It further constitutes four major components, which are as follows;

1. Inputs
2. Weights and Bias
3. Summation Functions
4. Activation or transformation function



The main logic behind the concept of Perceptron is as follows:

The inputs (x) are fed into the input layer, which undergoes multiplication with the allotted weights (w) followed by experiencing addition in order to form weighted sums. Then these inputs weighted sums with their corresponding weights are executed on the pertinent activation function.

Weights and Bias

As and when the input variable is fed into the network, a random value is given as a weight of that particular input, such that each individual weight represents the importance of that input in order to make correct predictions of the result.

However, bias helps in the adjustment of the curve of activation function so as to accomplish a precise output.

Summation Function

After the weights are assigned to the input, it then computes the product of each input and weights. Then the weighted sum is calculated by the summation function in which all of the products are added.

Activation Function

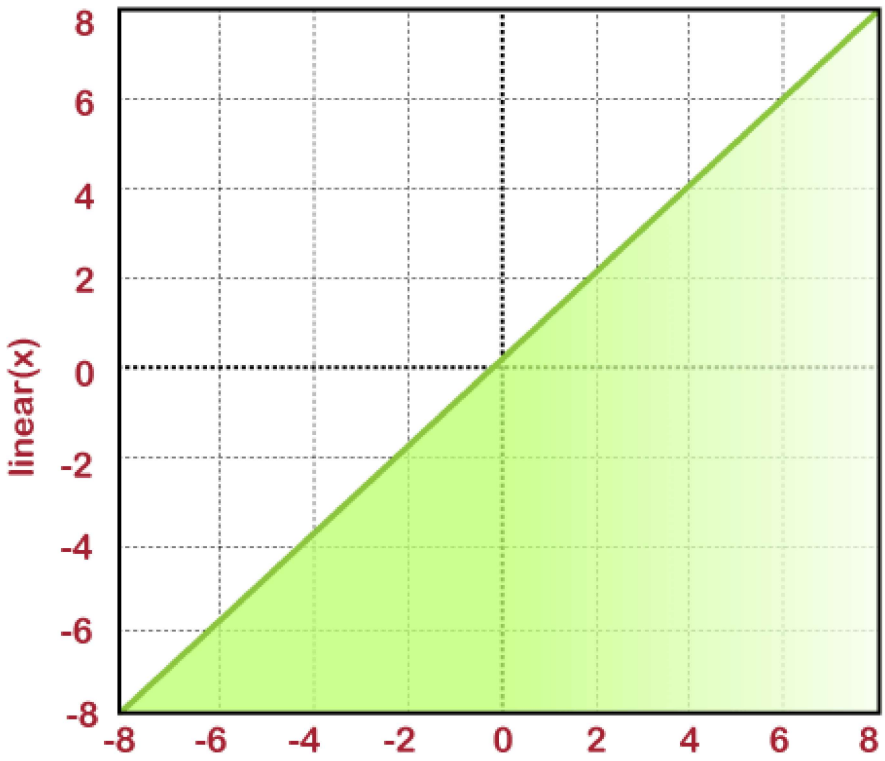
The main objective of the activation function is to perform a mapping of a weighted sum upon the output. The transformation function comprises of activation functions such as tanh, ReLU, sigmoid, etc.

The activation function is categorized into two main parts:

1. Linear Activation Function
2. Non-Linear Activation Function

Linear Activation Function

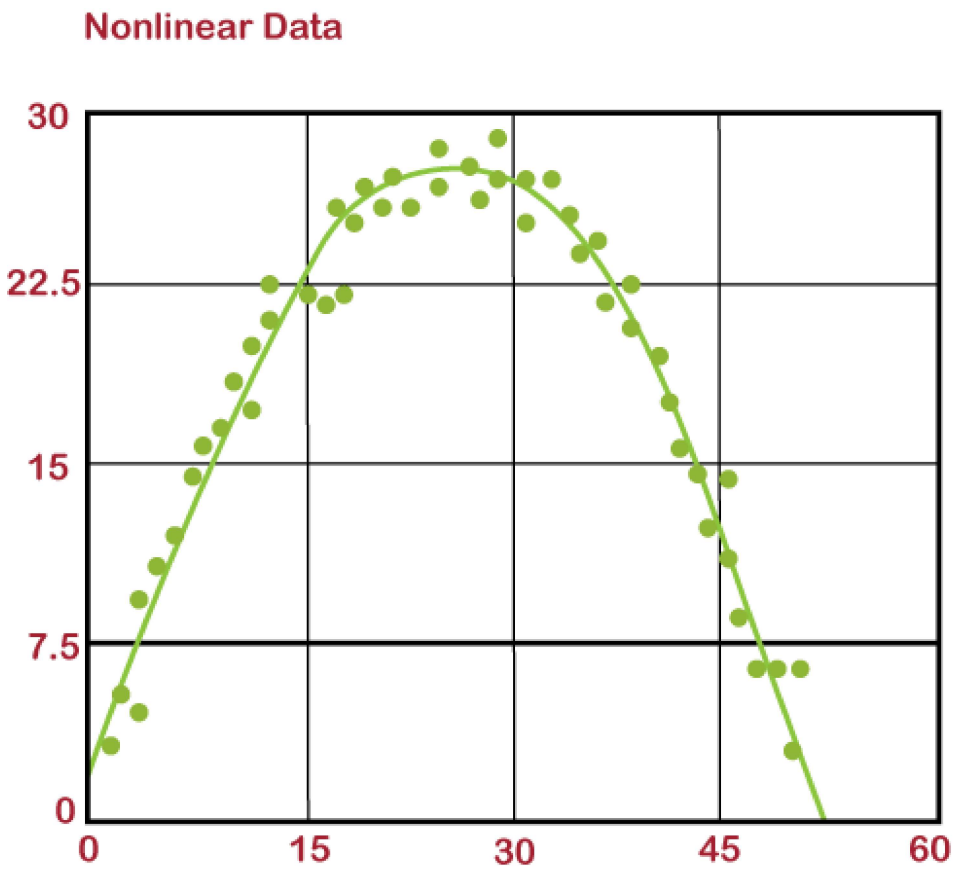
In the linear activation function, the output of functions is not restricted in between any range. Its range is specified from -infinity to infinity. For each individual neuron, the inputs get multiplied with the weight of each respective neuron, which in turn leads to the creation of output signal proportional to the input. If all the input layers are linear in nature, then the final activation of the last layer will actually be the linear function of the initial layer's input.



Non- linear function

These are one of the most widely used activation function. It helps the model in generalizing and adapting any sort of data in order to perform correct differentiation among the output. It solves the following problems faced by linear activation functions:

- Since the non-linear function comes up with derivative functions, so the problems related to backpropagation has been successfully solved.
- For the creation of deep neural networks, it permits the stacking up of several layers of the neurons.

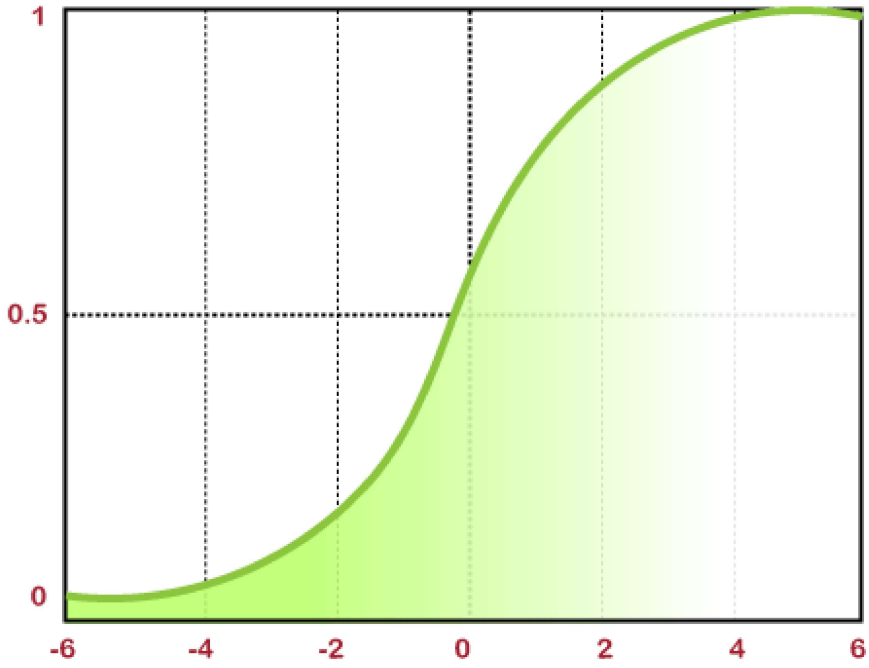


The non-linear activation function is further divided into the following parts:

1. **Sigmoid or Logistic Activation Function**

   It provides a smooth gradient by preventing sudden jumps in the output values. It has an output value range between 0 and 1 that helps in the normalization of each neuron's output. For X, if it has a value above 2 or below -2, then the values of y will be much steeper. In simple language, it means that even a small change in the X can bring a lot of change in Y.

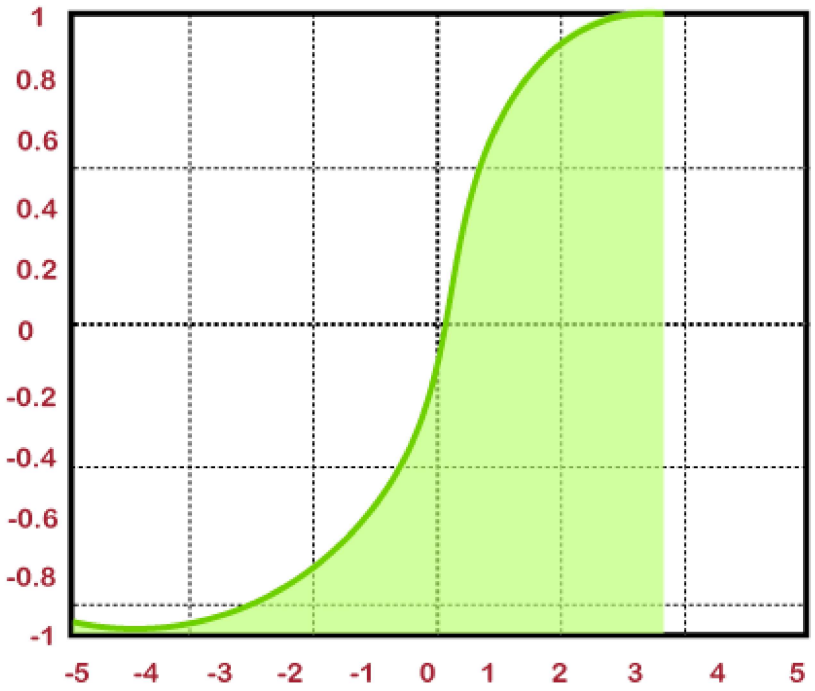
   It's value ranges between 0 and 1 due to which it is highly preferred by binary classification whose result is either 0 or 1.



2. **Tanh or Hyperbolic Tangent Activation Function**

   The tanh activation function works much better than that of the sigmoid function, or simply we can say it is an advanced version of the sigmoid activation function. Since it has a value range between -1 to 1, so it is utilized by the hidden layers in the neural network, and because of this

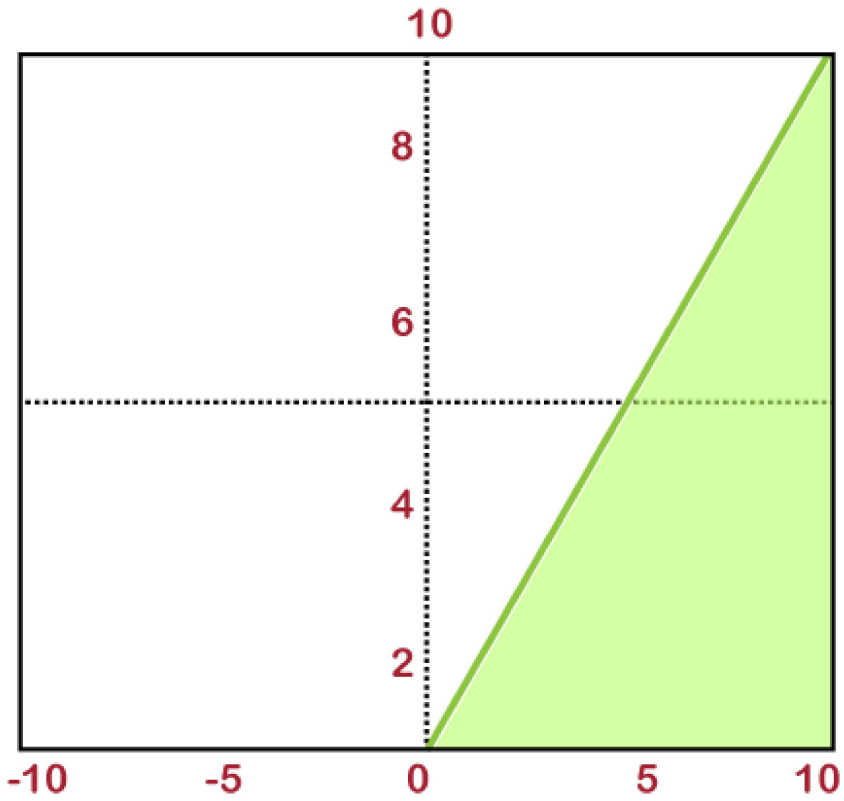
reason, it has made the process of learning much easier.



3. **ReLU(Rectified Linear Unit) Activation Function**

   ReLU is one of the most widely used activation function by the hidden layer in the neural network. Its value ranges from 0 to infinity. It clearly helps in solving out the problem of backpropagation. It tends out to be more expensive than the sigmoid, as well as the tanh activation function. It allows only a few neurons to get activated at a particular instance that

leads to effectual as well as easier computations.



4. **Softmax Function**
   It is one of a kind of sigmoid function whereby solving the problems of classifications. It is mainly used to handle multiple classes for which it squeezes the output of each class between 0 and 1, followed by dividing it by the sum of outputs. This kind of function is specially used by the classifier in the output layer.

Gradient Descent Algorithm

Gradient descent is an optimization algorithm that is utilized to minimize the cost function used in various machine learning algorithms so as to update the parameters of the learning model. In linear regression, these parameters are coefficients, whereas, in the neural network, they are weights.

Procedure:

It all starts with the coefficient's initial value or function's coefficient that may be either 0.0 or any small arbitrary value.

```
coefficient = 0.0
```

For estimating the cost of the coefficients, they are plugged into the function that helps in evaluating.

```
cost = f(coefficient)
or, cost = evaluate(f(coefficient))
```

Next, the derivate will be calculated, which is one of the concepts of calculus that relates to the function's slope at any given instance. In order to know the direction in which the values of the coefficient will move, we need to calculate the slope so as to accomplish a low cost in the next iteration.

```
delta = derivative(cost)
```

Now that we have found the downhill direction, it will further help in updating the values of coefficients. Next, we will need to specify alpha, which is a learning rate parameter, as it handles the amount of amendments made by coefficients on each update.

```
coefficient = coefficient - (alpha * delta)
```

Until the cost of the coefficient reaches **0.0** or somewhat close enough to it, the whole process will reiterate again and again.

It can be concluded that gradient descent is a very simple as well as straightforward concept. It just requires you to know about the gradient of the cost function or simply the function that you are willing to optimize.

Batch Gradient Descent

For every repetition of gradient descent, the main aim of batch gradient descent is to processes all of the training examples. In case we have a large number of training examples, then batch gradient descent tends out to be one of the most expensive and less preferable too.

Algorithm for Batch Gradient Descent

Let **m** be the number of training examples and **n** be the number of features.

Now assume that $\mathbf{h_\Theta}$ represents the hypothesis for linear regression and **∑** computes the sum of all training examples from **i=1 to m**. Then the cost of function will be computed by:

$$J_{train}(\Theta) = (1/2m) \sum (h_\Theta(x^{(i)}) - (y^{(i)})^2$$

Repeat {

$$\Theta j = \Theta j - (\text{learning rate}/m) * \sum (h_\Theta(x^{(i)}) - y^{(i)})\, x_j^{(i)}$$

For every j = 0...n

}

Here $x^{(i)}$ indicates the j^{th} feature of the i^{th} training example. In case if **m** is very large, then derivative will fail to converge at a **global minimum**.

Stochastic Gradient Descent

At a single repetition, the stochastic gradient descent processes only one training example, which means it necessitates for all the parameters to update after the one single training example is processed per single iteration. It tends to be much faster than that of the batch gradient descent, but when we have a huge number of training examples, then also it processes a single example due to which system may undergo a large no of repetitions. To evenly train the parameters provided by each type of data, properly shuffle the dataset.

Algorithm for Stochastic Gradient Descent

Suppose that $(x^{(i)}, y^{(i)})$ be the training example

Cost $(\Theta, (x^{(i)}, y^{(i)})) = (1/2) \sum (h\Theta(x^{(i)}) - (y^{(i)})^2$

$J_{train}(\Theta) = (1/m) \sum$ Cost $(\Theta, (x^{(i)}, y^{(i)}))$

Repeat {

For i=1 to m{

$\theta j = \theta j - (\text{learning rate}) * \sum (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

For every j=0...n

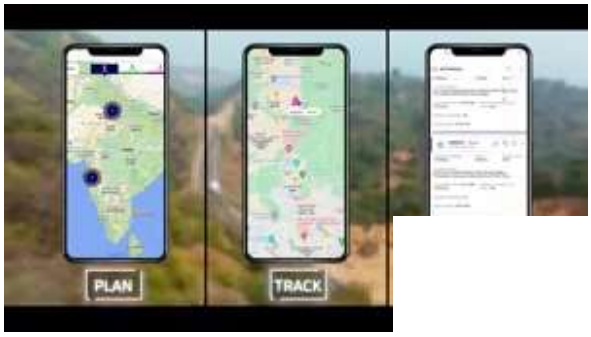
}

}

Convergence trends in different variants of Gradient Descent

The Batch Gradient Descent algorithm follows a straight-line path towards the minimum. The algorithm converges towards the **global minimum**, in case the cost function is **convex**, else towards the **local minimum**, if the cost function is not convex. Here the learning rate is typically constant.

However, in the case of Stochastic Gradient Descent, the algorithm fluctuates all over the global minimum rather than converging. The learning rate is changed slowly so that it can converge. Since it processes only one example in one iteration, it tends out to be noisy.



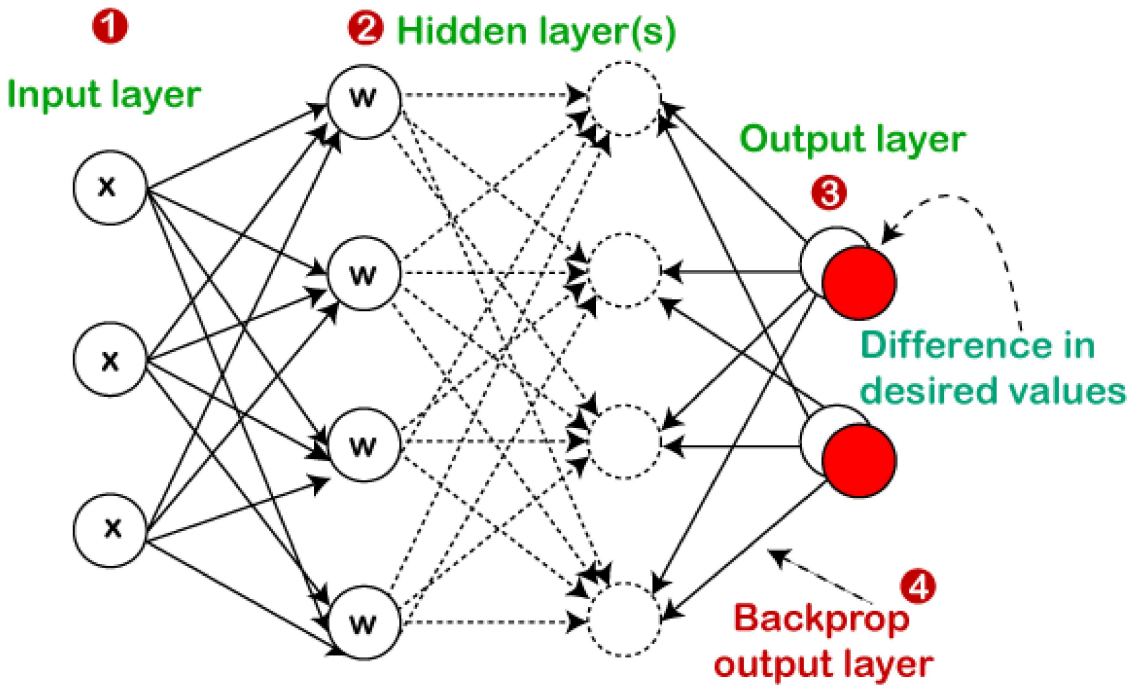
Backpropagation

The backpropagation consists of an input layer of neurons, an output layer, and at least one hidden layer. The neurons perform a weighted sum upon the input layer, which is then used by the activation function as an input, especially by the sigmoid activation function. It also makes use of supervised learning to teach the network. It constantly updates the weights of the network until the desired output is met by the network. It includes the following factors that are responsible for the training and performance of the network:

- Random (initial) values of weights.
- A number of training cycles.
- A number of hidden neurons.
- The training set.
- Teaching parameter values such as learning rate and momentum.

Working of Backpropagation

Consider the diagram given below.



1. The preconnected paths transfer the inputs **X**.
2. Then the weights **W** are randomly selected, which are used to model the input.
3. After then, the output is calculated for every individual neuron that passes from the input layer to the hidden layer and then to the output layer.
4. Lastly, the errors are evaluated in the outputs. **$Error_B$= Actual Output - Desired Output**
5. The errors are sent back to the hidden layer from the output layer for adjusting the weights to lessen the error.
6. Until the desired result is achieved, keep iterating all of the processes.

Need of Backpropagation

- Since it is fast as well as simple, it is very easy to implement.
- Apart from no of inputs, it does not encompass of any other parameter to perform tuning.
- As it does not necessitate any kind of prior knowledge, so it tends out to be more flexible.
- It is a standard method that results well.

Building an ANN

Before starting with building an ANN model, we will require a dataset on which our model is going to work. The dataset is the collection of data for a particular problem, which is in the form of a CSV file.

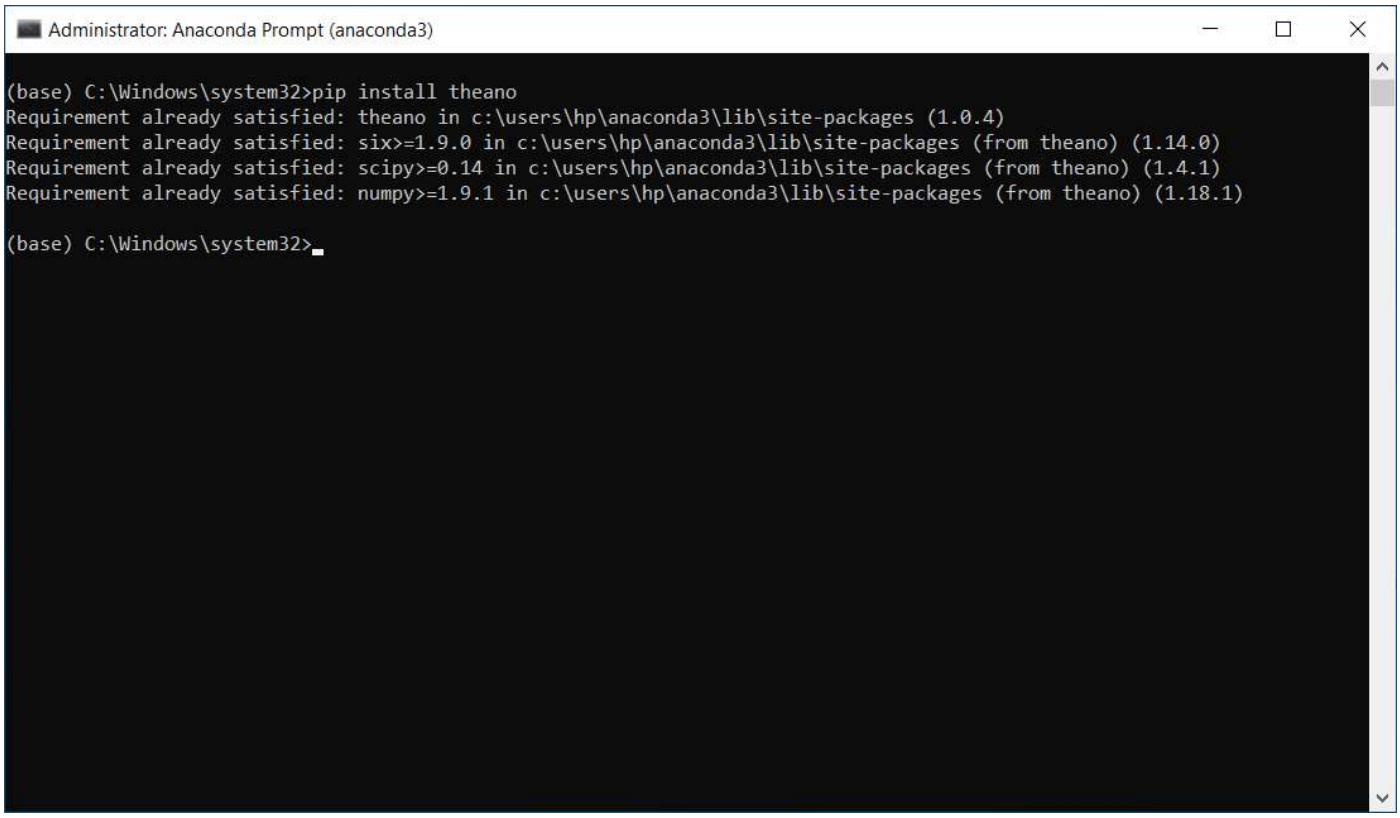
CSV stands for **Comma-separated values** that save the data in the tabular format. We are using a fictional dataset of banks. The bank dataset contains data of its 10,000 customers with their details. This whole thing is undergone because the bank is seeing some unusual churn rates, which is nothing but the customers are leaving at an unusual high rate, and they want to know the reason behind it so that they can assess and address that particular problem.

Here we are going to solve this business problem using artificial neural networks. The problem that we are going to deal with is a **classification problem**. We have several independent variables like Credit Score, Balance, and Number of Products on the basis of which we are going to predict which customers are leaving the bank. Basically, we are going to do a classification problem, and artificial neural networks can do a terrific job at making such kind of predictions.

So, we will start with installing the **Keras** library, **TensorFlow** library, as well as the **Theano** library on Anaconda Prompt, and for that, you need to open it as administrator followed by running the commands one after other as given below.

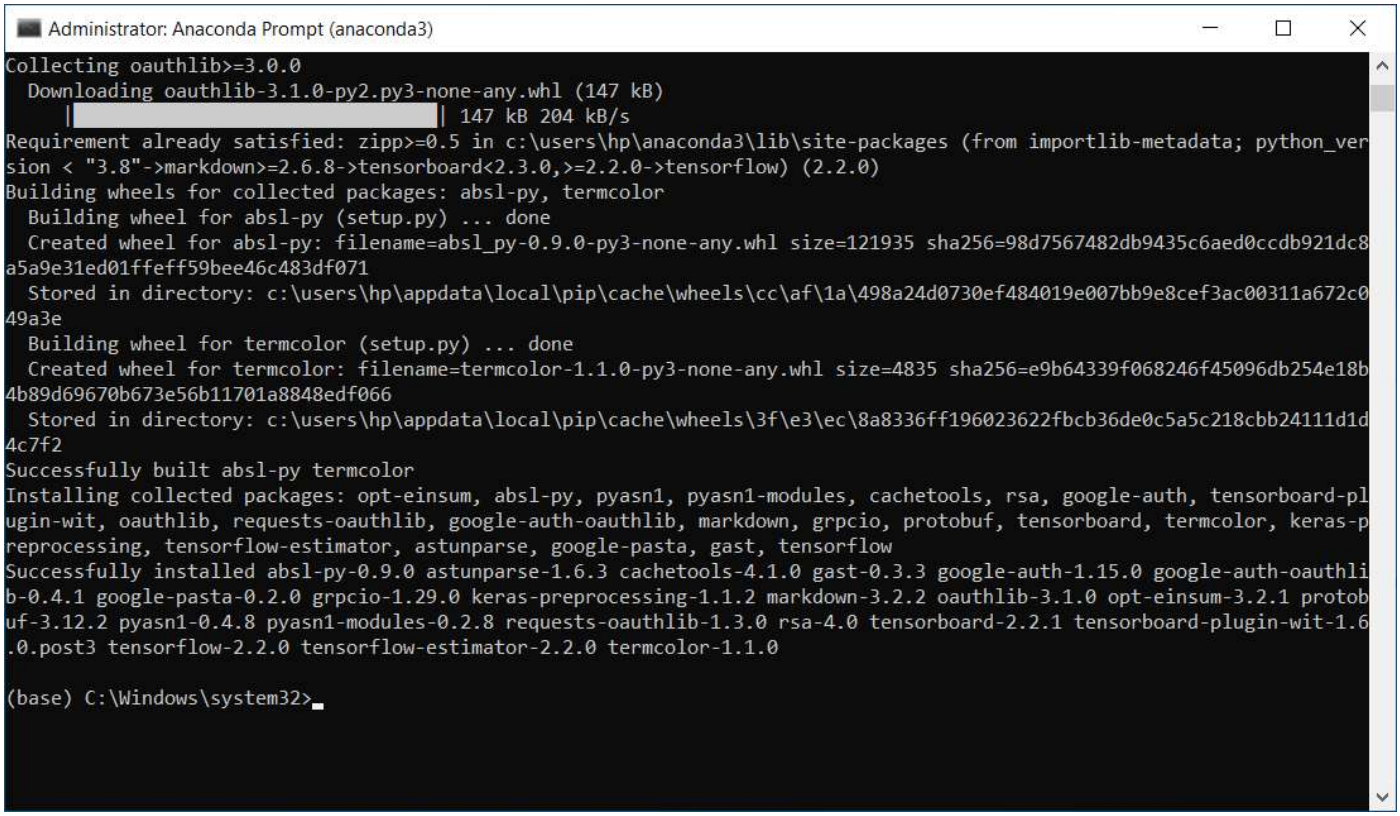
```
pip install theano
```

Since it is already installed, the output will be as given below.

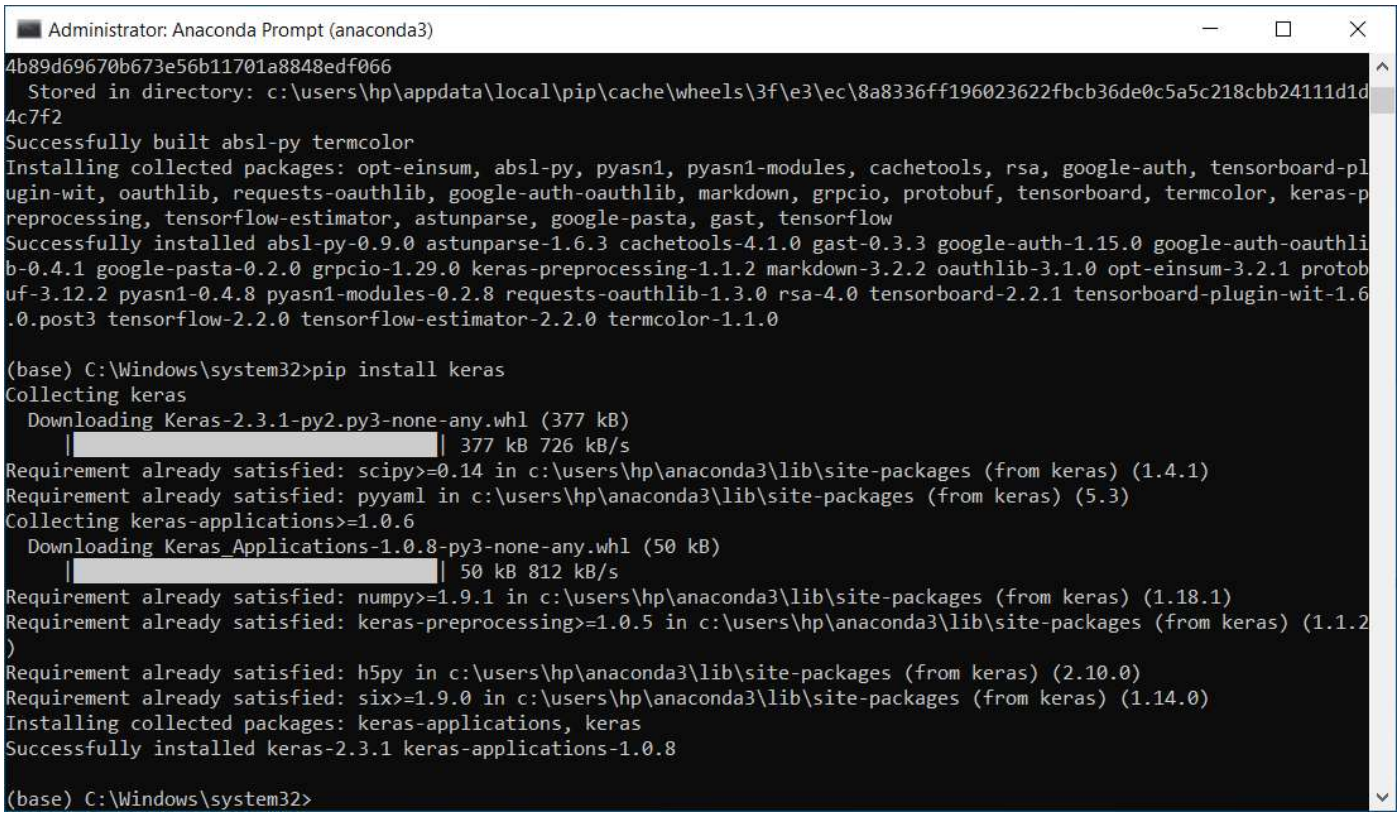



```
pip install tensorflow
```

From the image given below, it can be seen that the TensorFlow library is successfully installed.



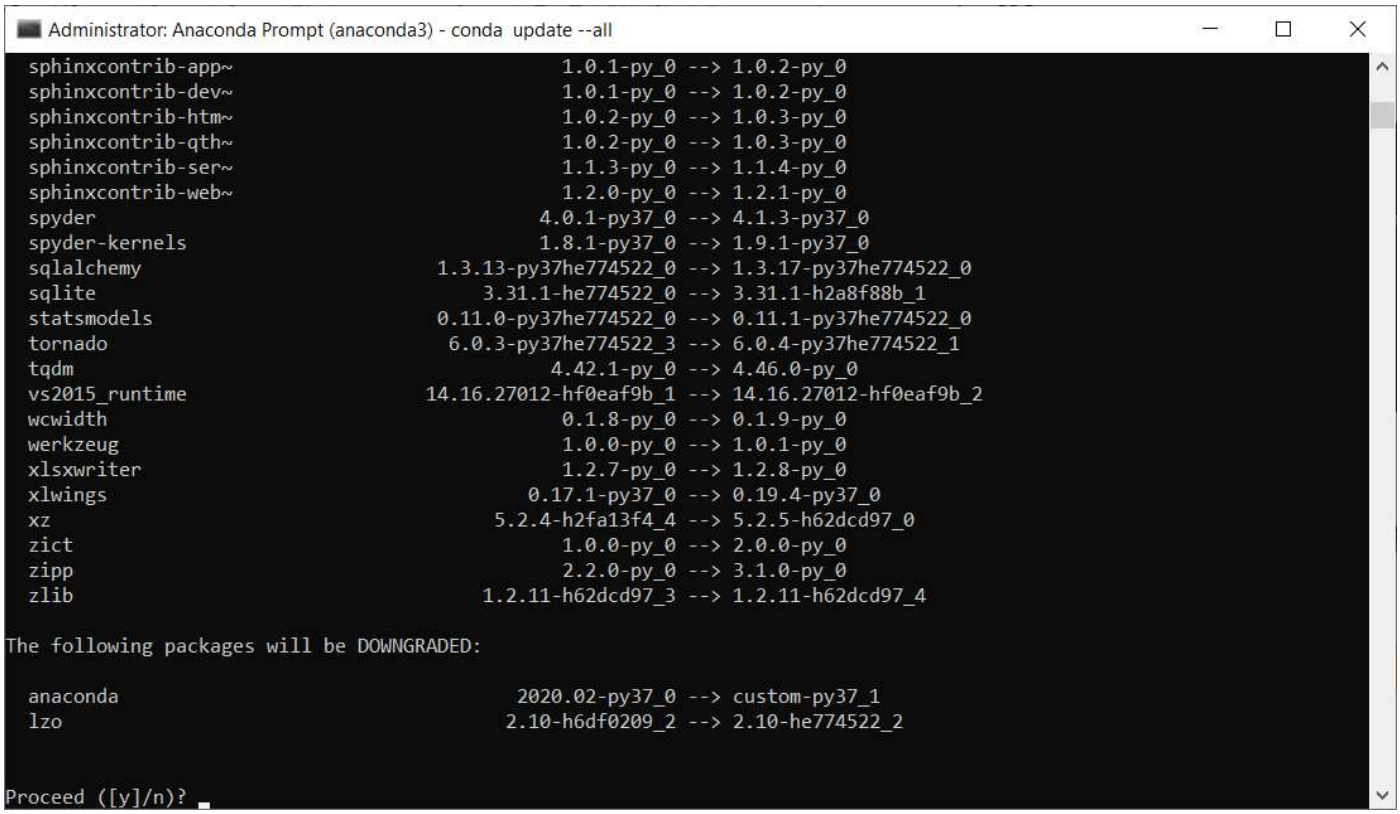
pip install keras



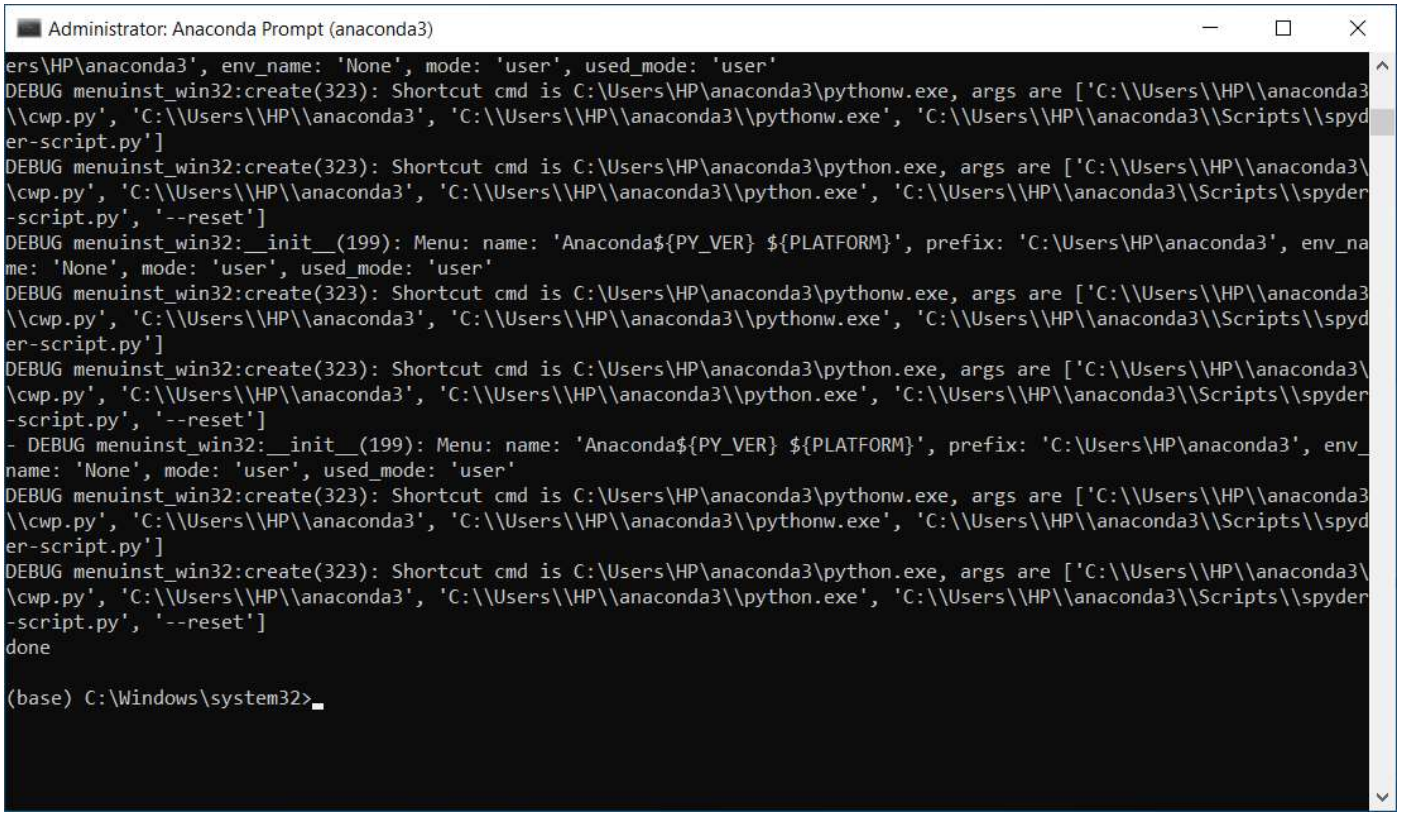
So, we have installed Keras library too.

Now that we are done with the installation, the next step is to update all these libraries to the most updated version, and it can be done by following the given code.

```
conda update --all
```



Since we are doing it for the very first time, it will ask whether to proceed or not. Confirm it with y and press enter.



After the libraries are updated successfully, we will close the Anaconda prompt and get back to the Spyder IDE.

Now we will start building our model in two parts, such that in part **1st**, we will do **data pre-processing**, however in **2nd** part, we will **create the ANN model**.

Data pre-processing is very necessary to prepare the data correctly for building a future deep learning model. Since we are in front of a classification problem, so we have some independent variables encompassing some information about customers in a bank, and we are trying to predict the binary outcome for the dependent variable, i.e., either **1** if the customer leaves the bank or **0** if the customer stays in the bank.

Part1: Data Pre-processing

We will start by importing some of the pre-defined Python libraries such as NumPy, Matplotlib, and Pandas so as to perform data-preprocessing. All these libraries perform some sort of specific tasks.

NumPy

NumPy is a python library that stands for **Numerical Python**, allows the implementation of linear, mathematical and logical operations on arrays as well as Fourier transformation and routine to manipulate the shapes.

```
import numpy as np
```

Matplotlib

It is also an open-source library with the help of which charts can be plotted in the python. The sole purpose of this library is to visualize the data for which it necessitates to import its **pyplot** sub library.

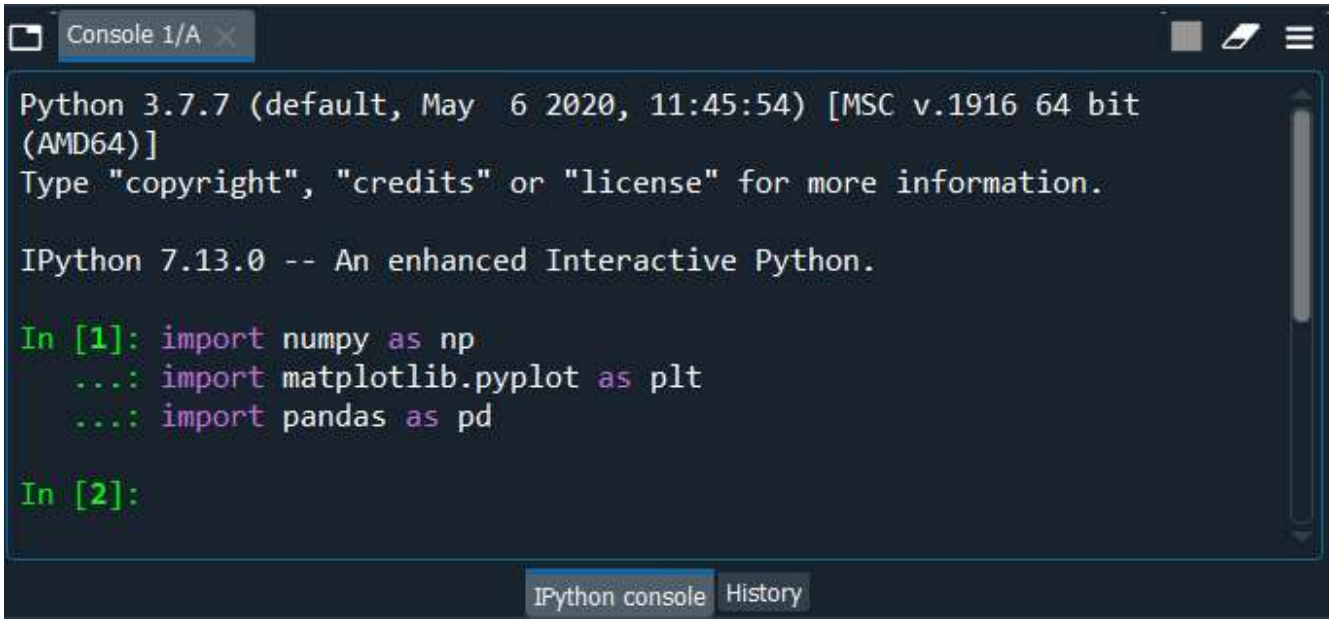
```
import matplotlib.pyplot as plt
```

Pandas

Pandas is also an open-source library that enables high-performance data manipulation as well as analyzing tools. It is mainly used to handle the data and make the analysis.

```
import pandas as pd
```

An output image is given below, which shows that the libraries have been successfully imported.

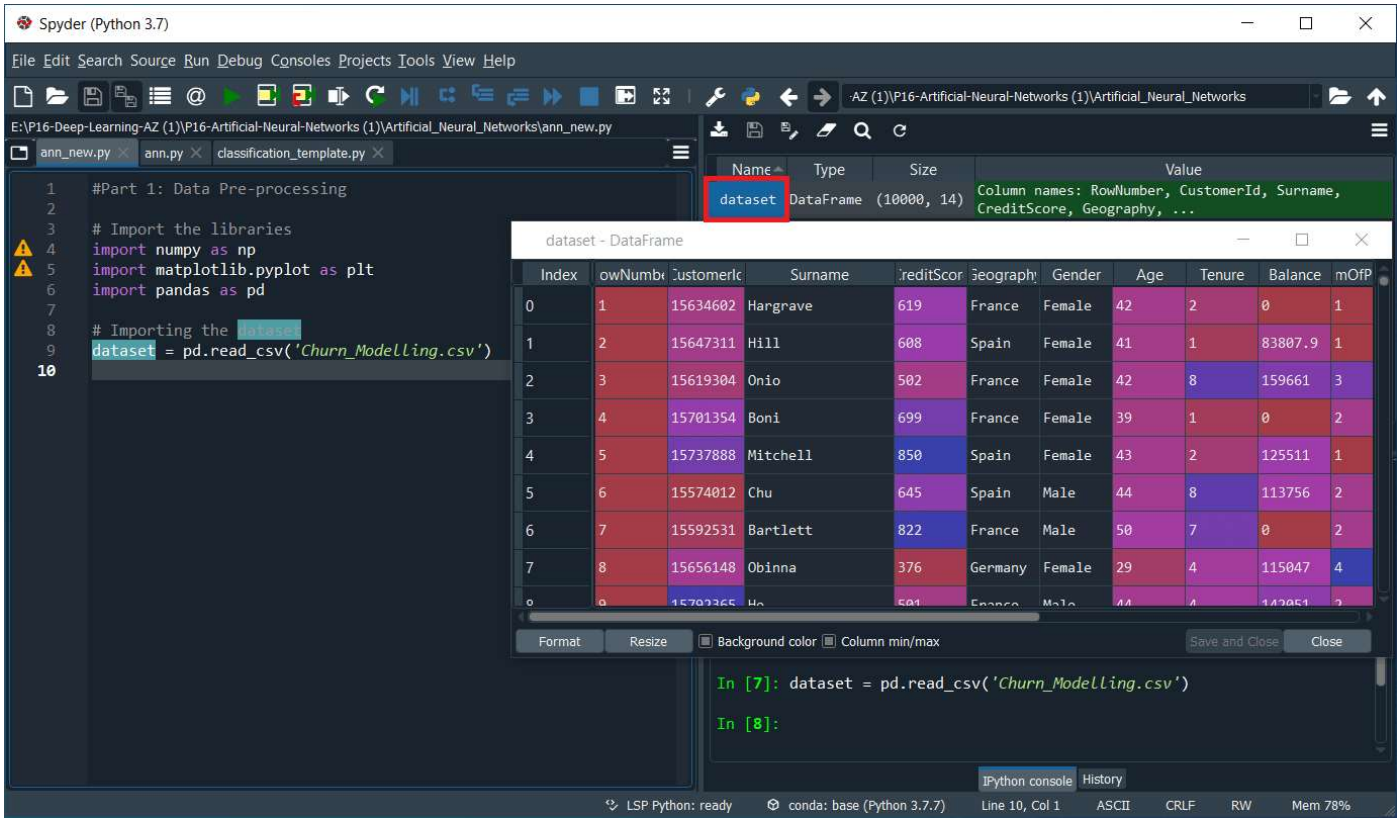


Next, we will import the data file from the current working directories with the help of Pandas. We will use **read.csv()** for reading the CSV file both locally as well as through the URL.

```
dataset = pd.read_csv('Churn_Modelling.csv')
```

From the code given above, **the dataset** is the name of the variable in which we are going to save the data. We have passed the name of the dataset in the **read.csv()**. Once the code is run, we can see that the data is uploaded successfully.

By clicking on the **Variable explorer** and selecting **the dataset**, we can check the dataset, as shown in the following image.



Next, we will create the **matrix of feature**, which is nothing but a matrix of the independent variable. Since we don't know which independent variable might has the most impact on the dependent variable, so that is what our artificial neural network will spot by looking at the correlations; it will give bigger weight to those independent variables that have the most impact in the neural network.

So, we will include all the independent variables from the **credit score** to the last one that is the estimated salary.

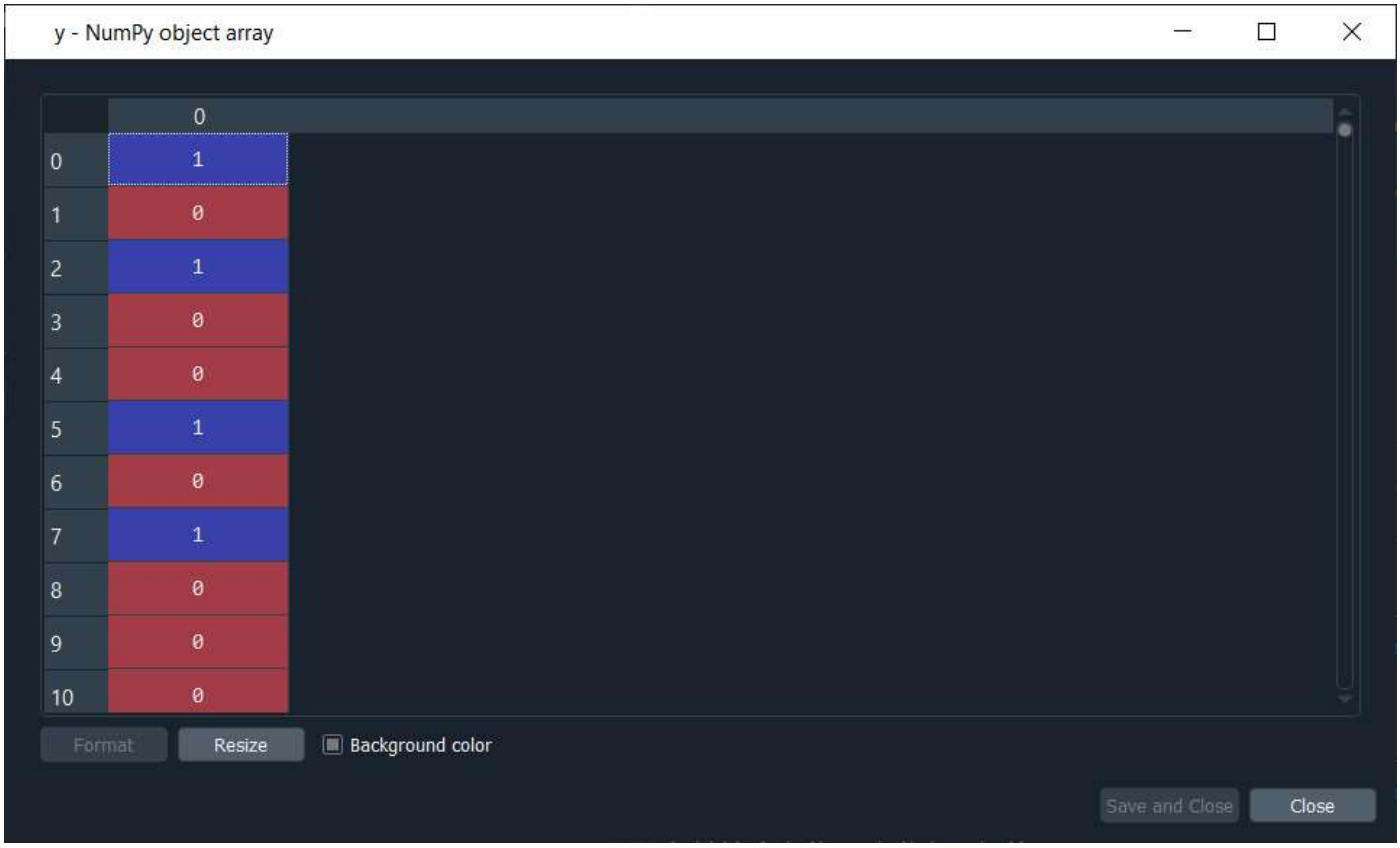
```
X = dataset.iloc[:, 3:13].values
```

After running the above code, we will see that we have successfully created the matrix of feature **X**. Next, we will create a **dependent variable vector**.

```
y = dataset.iloc[:, 13].values
```

By clicking on **y**, we can have a look that y contains **binary outcome**, i.e., 0 or 1 for all the 10,000 customers of the bank.

Output:



Next, we will split the dataset into the training and test set. But before that, we need to encode that matrix of the feature as it contains the **categorical data**. Since the dependent variable also comprises of categorical data but sidewise, it also takes a numerical value, so don't need to encode text into numbers. But then again, we have our independent variable, which has categories of strings, so we need to encode the categorical independent variables.

The main reason behind encoding the categorical data before splitting is that it is must to encode the matrix of **X** and the dependent variable **y**.

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X = LabelEncoder()
X[:, 0] = labelencoder_X.fit_transform(X[:, 0])
onehotencoder = OneHotEncoder(categorical_features = [0])
X = onehotencoder.fit_transform(X).toarray()
```

So, now we will encode our categorical independent variable by having a look at our matrix from console and for that we just need to press **X** at the console.

Output:

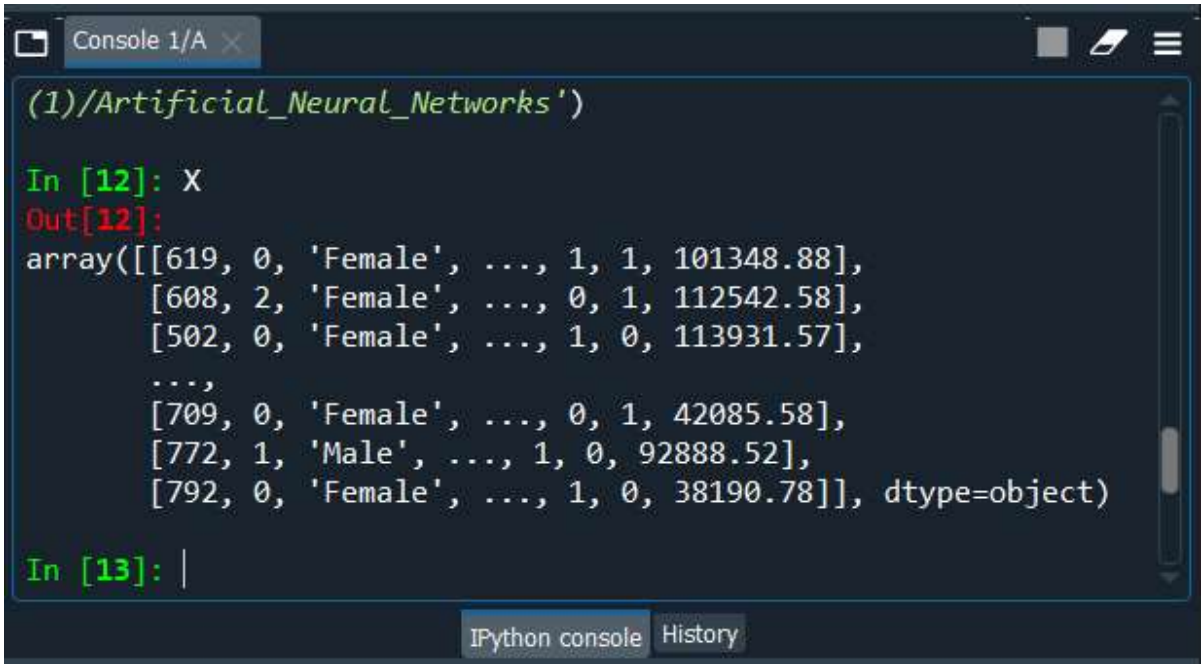
From the image given above, we can see that we have only two categorical independent variables, which is the **country variable** containing three countries, i.e., France, Spain, and Germany, and the other one is the **gender variable**, i.e., male and female. So, we have got these two variables, which we will encode in our matrix of features.

So we will need to create two label encoder objects, such that we will create our first label encoder object named **labelencoder_X_1** followed by applying **fit_transform** method to encode this variable, which will, in turn, the strings here France, Spain, and Germany into the numbers 0, 1 and 2.

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X_1 = LabelEncoder()
X[:, 1] = labelencoder_X_1.fit_transform(X[:, 1])
```

After executing the code, we will now have a look at the **X** variable, simply by pressing X in the console, as we did in the earlier step.

Output:

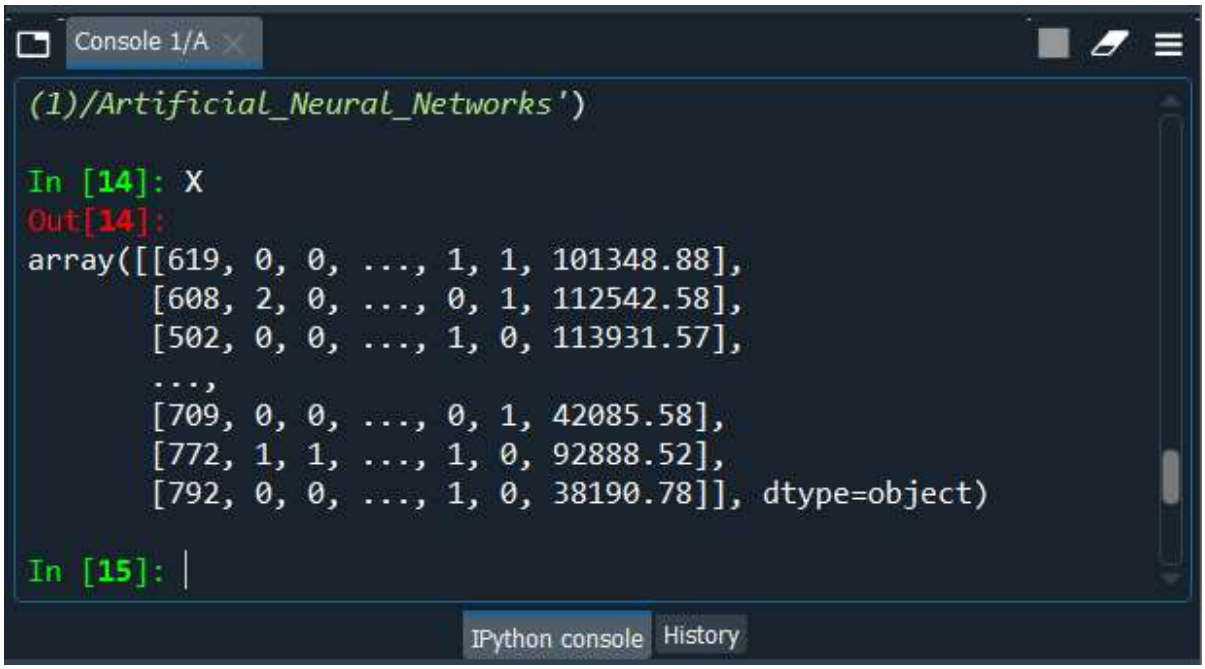


So, from the output image given above, we can see that France became 0, Germany became 1, and Spain became 2.

Now in a similar manner, we will do the same for the other variable, i.e., Gender variable but with a new object.

```
labelencoder_X_2 = LabelEncoder()
X[:, 2] = labelencoder_X_2.fit_transform(X[:, 2])
```

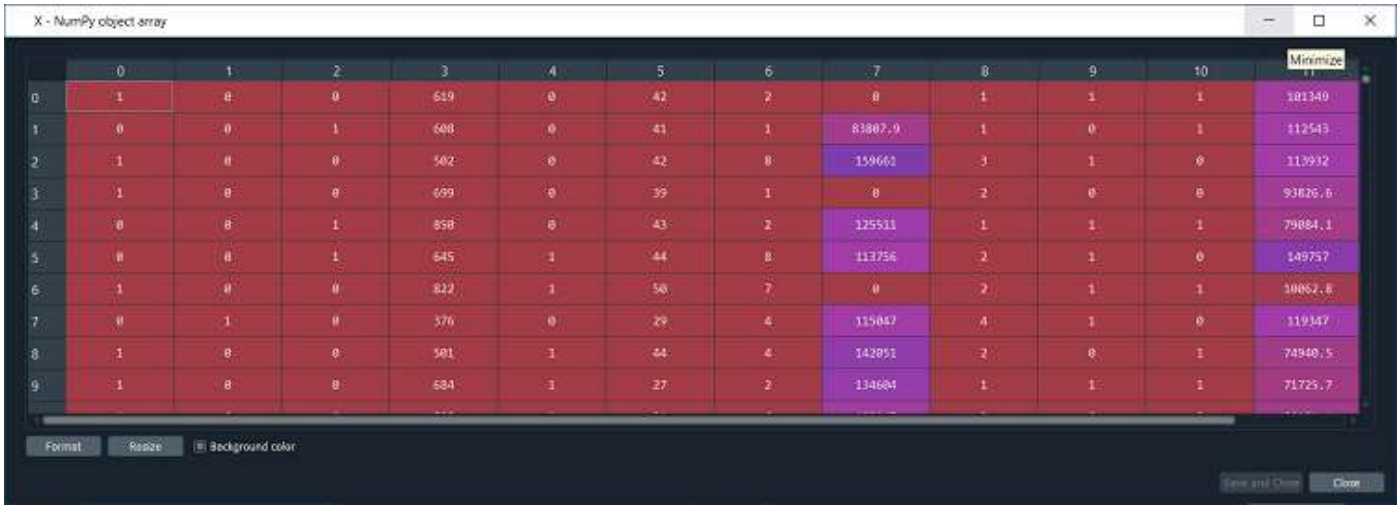
Output:



We can clearly see that females became 0 and males became 1. Since there is no relational order between the categories of our categorical variable, so for that we need to create a dummy variable for the country categorical variable as it contains three categories unlike the gender variable having only two categories, which is why we will be removing one column to avoid the **dummy variable trap**. It is useless to create the dummy variable for the gender variable. We will use **the OneHotEncoder** class to create the dummy variables.

```
from sklearn.compose import ColumnTransformer
label_encoder_x_1 = LabelEncoder()
X[: , 2] = label_encoder_x_1.fit_transform(X[:,2])
transformer = ColumnTransformer(
    transformers=[
        ("OneHot",        # Just a name
         OneHotEncoder(), # The transformer class
         [1]              # The column(s) to be applied on.
         )
    ],
    remainder='passthrough' # don't apply anything to the remaining columns
)
X = transformer.fit_transform(X.tolist())
X = X.astype('float64')
```

Output:

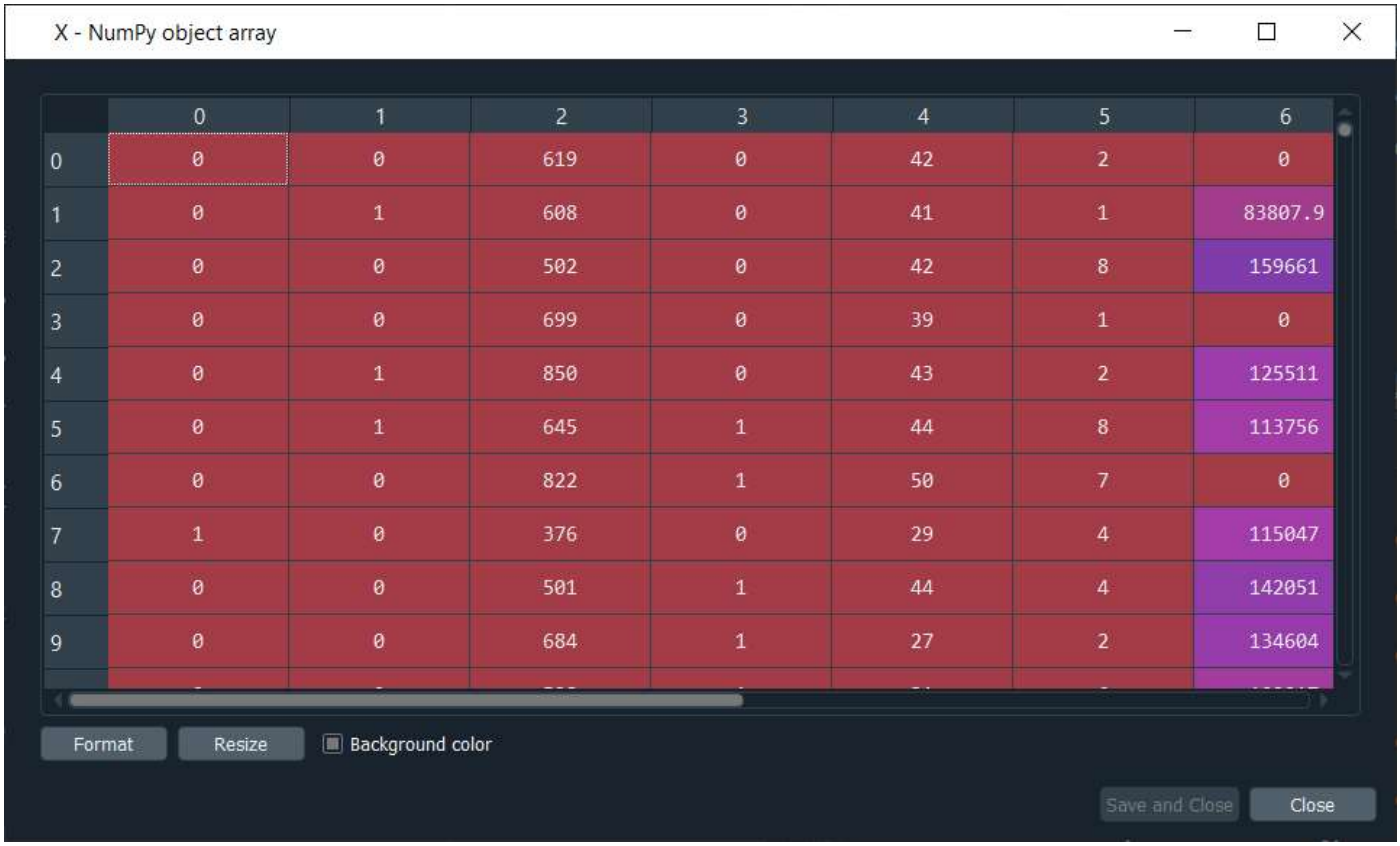


By having a look at **X**, we can see that all the columns are of the same type now. Also, the type is no longer an object but float64. We can see that we have twelve independent variables because we have three new dummy variables.

Next, we will remove one dummy variable to avoid falling into a dummy variable trap. We will take a matrix of features X and update it by taking all the lines of this matrix and all the columns except the first one.

```
X = X[:, 1:]
```

Output:



	0	1	2	3	4	5	6
0	0	0	619	0	42	2	0
1	0	1	608	0	41	1	83807.9
2	0	0	502	0	42	8	159661
3	0	0	699	0	39	1	0
4	0	1	850	0	43	2	125511
5	0	1	645	1	44	8	113756
6	0	0	822	1	50	7	0
7	1	0	376	0	29	4	115047
8	0	0	501	1	44	4	142051
9	0	0	684	1	27	2	134604

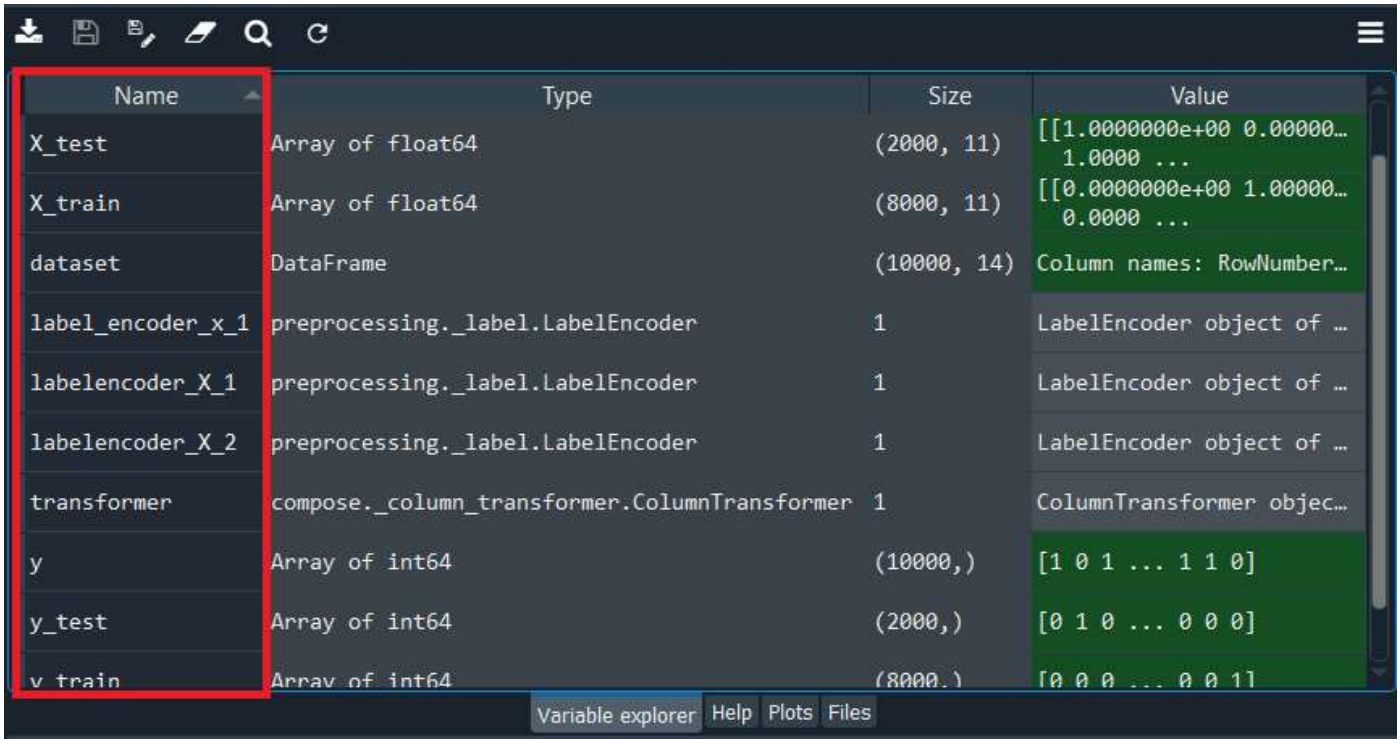
It can be seen that we are left with only two dummy variables, so no more dummy variable trap.

Now we are ready to split the dataset into the training set and test set. We have taken the **test size** to **0.2** for training the ANN on **8,000** observations and testing its performance on **2,000** observations.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

By executing the code given above, we will get four different variables that can be seen under the variable explorer section.

Output:



Name	Type	Size	Value
X_test	Array of float64	(2000, 11)	[[1.0000000e+00 0.00000… 1.0000 ...
X_train	Array of float64	(8000, 11)	[[0.0000000e+00 1.00000… 0.0000 ...
dataset	DataFrame	(10000, 14)	Column names: RowNumber…
label_encoder_x_1	preprocessing._label.LabelEncoder	1	LabelEncoder object of …
labelencoder_X_1	preprocessing._label.LabelEncoder	1	LabelEncoder object of …
labelencoder_X_2	preprocessing._label.LabelEncoder	1	LabelEncoder object of …
transformer	compose._column_transformer.ColumnTransformer	1	ColumnTransformer objec…
y	Array of int64	(10000,)	[1 0 1 ... 1 1 0]
y_test	Array of int64	(2000,)	[0 1 0 ... 0 0 0]
y_train	Array of int64	(8000,)	[0 0 0 ... 0 0 1]

Variable explorer | Help | Plots | Files

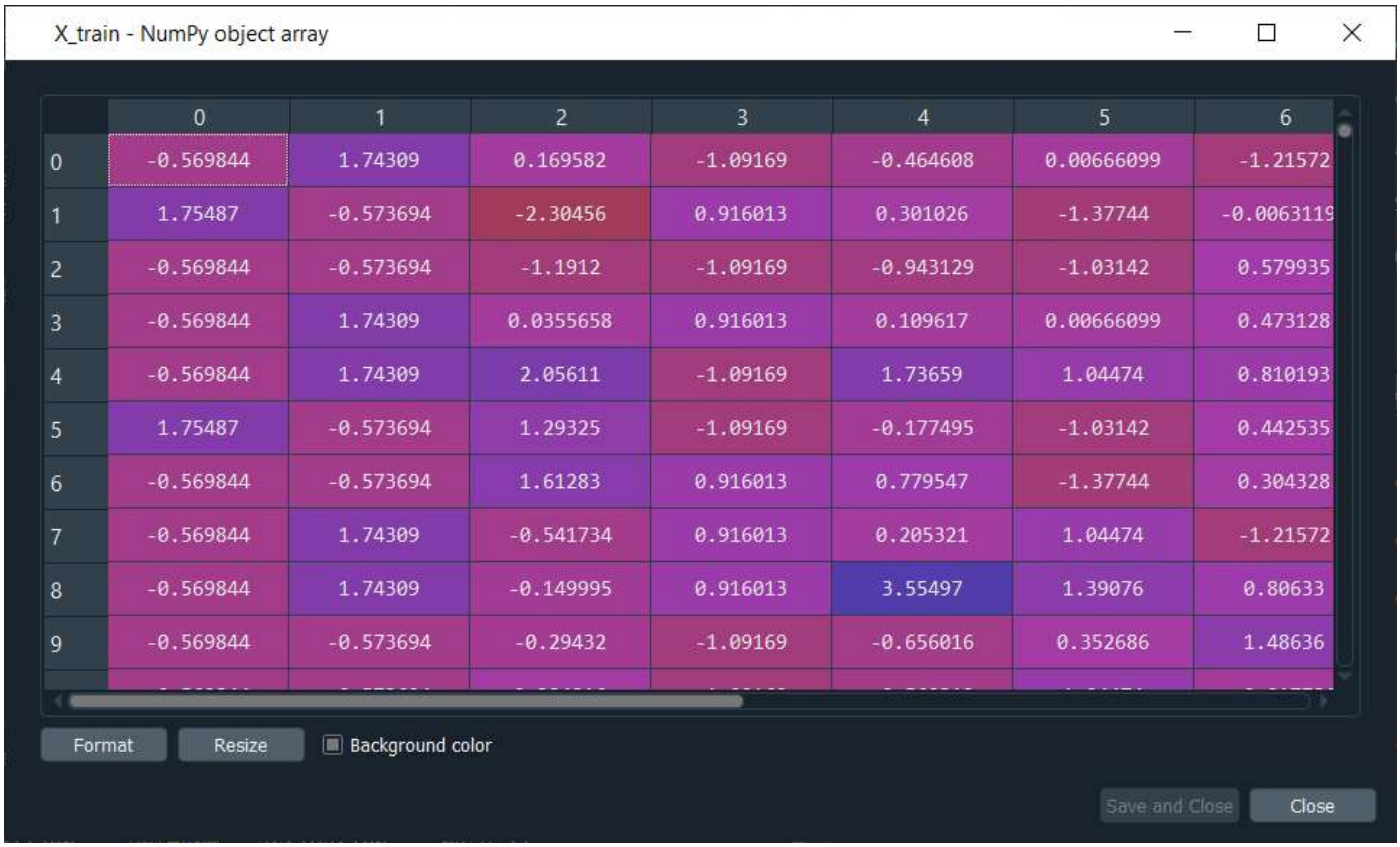
Besides parallel computations, we are going to have highly computed intensive calculations as well as we don't want one independent variable dominating the other one, so we will be applying feature scaling to ease out all the calculations.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

After executing the above code, we can have a quick look at **X_train** and **X_test** to check if all the independent variables are scaled properly or not.

Output:

X_train



	0	1	2	3	4	5	6
0	-0.569844	1.74309	0.169582	-1.09169	-0.464608	0.00666099	-1.21572
1	1.75487	-0.573694	-2.30456	0.916013	0.301026	-1.37744	-0.0063119
2	-0.569844	-0.573694	-1.1912	-1.09169	-0.943129	-1.03142	0.579935
3	-0.569844	1.74309	0.0355658	0.916013	0.109617	0.00666099	0.473128
4	-0.569844	1.74309	2.05611	-1.09169	1.73659	1.04474	0.810193
5	1.75487	-0.573694	1.29325	-1.09169	-0.177495	-1.03142	0.442535
6	-0.569844	-0.573694	1.61283	0.916013	0.779547	-1.37744	0.304328
7	-0.569844	1.74309	-0.541734	0.916013	0.205321	1.04474	-1.21572
8	-0.569844	1.74309	-0.149995	0.916013	3.55497	1.39076	0.80633
9	-0.569844	-0.573694	-0.29432	-1.09169	-0.656016	0.352686	1.48636

X_test

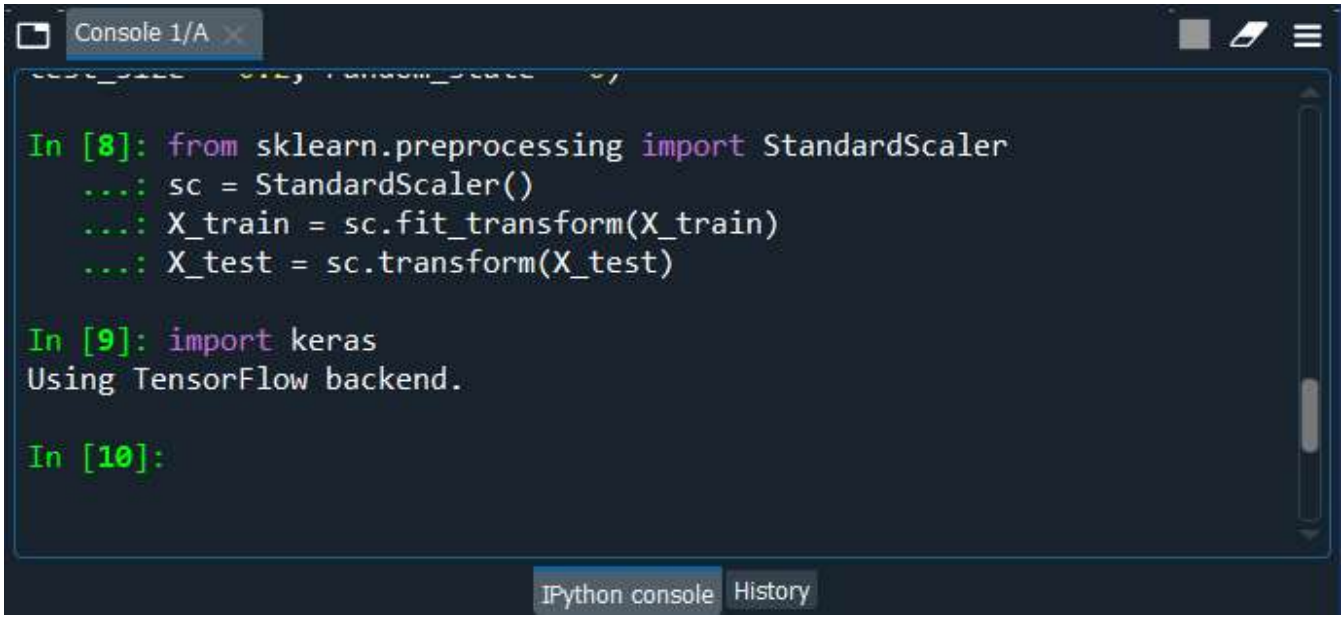
	0	1	2	3	4	5	6
0	1.75487	-0.573694	-0.552043	-1.09169	-0.368904	1.04474	0.879303
1	-0.569844	-0.573694	-1.3149	-1.09169	0.109617	-1.03142	0.429722
2	-0.569844	1.74309	0.57163	-1.09169	0.301026	1.04474	0.308583
3	-0.569844	-0.573694	1.41696	0.916013	-0.656016	-0.339364	0.575336
4	1.75487	-0.573694	0.57163	0.916013	-0.0817912	0.00666099	1.38961
5	-0.569844	1.74309	0.200509	-1.09169	1.73659	-0.68539	1.59002
6	-0.569844	1.74309	-0.624205	0.916013	-0.464608	-1.72347	-0.164023
7	-0.569844	1.74309	-0.149995	-1.09169	-0.943129	0.352686	1.30239
8	-0.569844	-0.573694	-0.541734	-1.09169	2.40652	1.39076	-1.21572
9	-0.569844	-0.573694	-2.0056	-1.09169	2.31081	-1.37744	1.42662

Now that our data is well pre-processed, we will start by building an artificial neural network.

Part2: Building an ANN

We will start with importing the Keras libraries as well as the desired packages as it will build the Neural Network based on TensorFlow

```
import keras
```



After importing the Keras library, we will now import two modules, i.e., the Sequential module, which is required to initialize our neural network, and the Dense module that is needed to build the layer of our ANN.

```
from keras.models import Sequential
from keras.layers import Dense
```

Next, we will initialize the ANN, or simply we can say we will be defining it as a sequence of layers. The deep learning model can be initialized in two ways, either by defining the sequence of layers or defining a graph. Since we are going to make our ANN with successive layers, so we will initialize our deep learning model by defining it as a sequence of layers.

It can be done by creating an object of the sequential class, which is taken from the sequential model. The object that we are going to create is nothing but the model itself, i.e., a neural network that will have a row of classifiers because we are solving a classification problem where we have to predict a class, so our neural network model is going to be a classifier. As in the next step, we will be predicting the test set result using the classifier name, so we will call our model as a classifier that is nothing but our future Artificial Neural Network that we are going to build.

Since this classifier is an object of Sequential class, so we will be using it, but will not pass any argument because we will be defining the layers step by step by starting with the input layer followed by adding some hidden layers and then the output layer.

```
classifier = Sequential()
```

After this, we will start by adding the input layer and the first hidden layer. We will take the classifier that we initialized in the previous step by creating an object of the sequential class, and we will use the **add()** method to add different layers in our neural network. In the add(), we will pass the **layer** argument, and since we are going to add two layers, i.e., the input and first hidden layer, which we will be doing with the help of **Dense()** function that we have mentioned above.

Within the **Dense()** function we will pass the following arguments;

- **units** are the very first argument, which can be defined as the number of nodes that we want to add in the hidden layer.
- The second argument is the **kernel_initializer** that randomly initializes the weight as a small number close to zero so that they can be randomly initialized with a uniform function. Here we have a simple **uniform** function that will initialize the weight according to the uniform distribution.
- The third argument is the **activation**, which can be understood as the function that we want to choose in our hidden layer. So, we will be using the **rectifier function** for the **hidden layers** and the **sigmoid function** for the **output layer**. Since we are in the hidden layer, we are using the "**relu**" perimeter as it corresponds to the rectifier function.
- And the last is the **input_dim** argument that specifies the number of nodes in the input layer, which is actually the number of independent variables. It is very necessary to add the argument because, by so far, we have only initialized our ANN, we haven't created any layer yet, and that's why it doesn't know which node this hidden layer we are creating is expecting as inputs. After the first hidden layer gets created, we don't need to specify this argument for the next hidden layers.

```
classifier.add(Dense(units = 6, kernel_initializer = 'uniform', activation = 'relu', input_dim = 11))
```

Next, we will add the second hidden layer by using the same add method followed by passing the same parameter, which is the **Dense()** as well as the same parameters inside it as we did in the earlier step except for the **input_dim**.

```
classifier.add(Dense(units = 6, kernel_initializer = 'uniform', activation = 'relu'))
```

After adding the two hidden layers, we will now add the final output layer. This is again similar to the previous step, just the fact that we will be units parameter because in the output layer we only require one node as our dependent variable is a categorical variable encompassing a binary outcome and also when we have binary outcome then, in that case, we have only one node in the output layer. So, therefore, we will put units equals to 1, and since we are in the output layer, we will be replacing the **rectifier** function to **sigmoid** activation function.

```
classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation = 'sigmoid'))
```

As we are done with adding the layers of our ANN, we will now compile the whole artificial neural network by applying the stochastic gradient descent. We will start with our classifier object, followed by using the compile method and will pass on the following arguments in it.

- The first argument is the **optimizer**, which is simply the algorithm that we want to use to find the optimal set of weights in the neural networks. The algorithm that we are going to use is nothing but the stochastic gradient descent algorithm. Since there are several types of stochastic descent algorithms and the most efficient one is called "**adam**," which is going to be the input of this optimizer parameter.
- The second parameter is the loss, which is a loss function within the stochastic gradient descent algorithm, which is used to find the optimal weights. Since our dependent variable has a **binary outcome**, so we will be using **binary_crossentropy** logarithmic function, and when there is a **binary outcome**, then we will incorporate **categorical_crossentropy**.
- The last argument will be the metrics, which is nothing but a criterion to evaluate our model, and we are using the "**accuracy**." So, what happens is when the weights are updated after each observation, the algorithm makes use of this accuracy to improve the model's performance.

```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

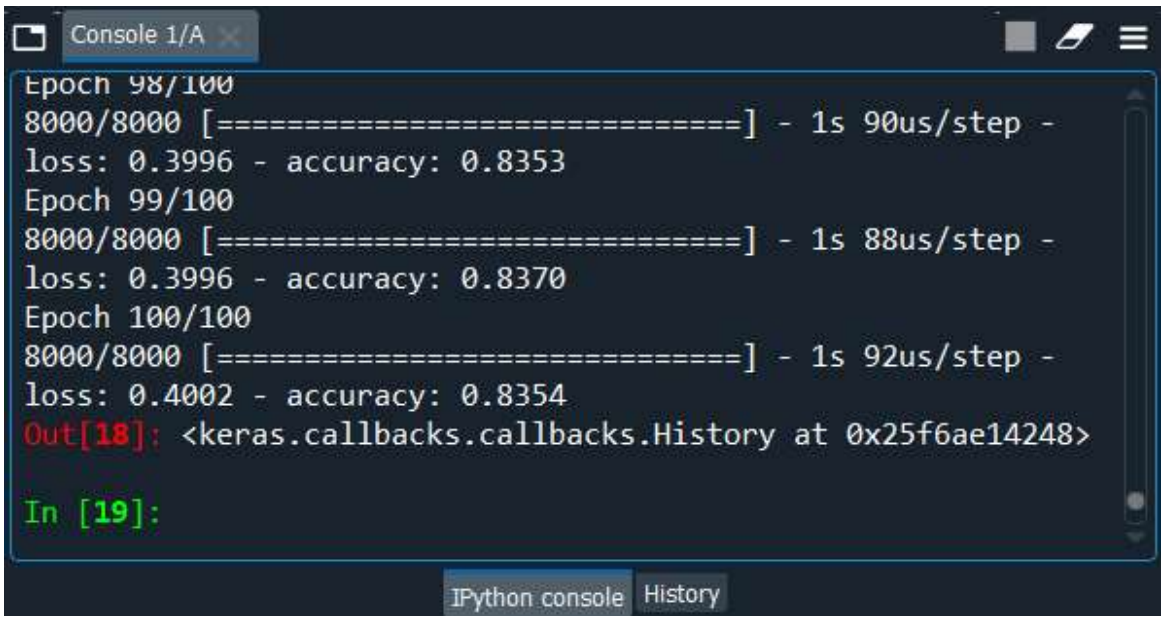
Next, we will fit the ANN to the training set for which we will be using the fit method to fit our ANN to the training set. In the fit method, we will be passing the following arguments:

- The first argument is the dataset on which we want to train our classifier, which is the training set separated into two-argument such as **X_train** (matrix of feature containing the observations of the train set) and **y_train** (containing the actual outcomes of the dependent variable for all the observations in the training set).

- The next argument is the **batch_size**, which is the number of observations, after which we want to update the weight.
- And lastly, the no. of **epochs** that we are going to apply to see the algorithm in action as well the improvement in accuracy over the different epochs.

```
classifier.fit(X_train, y_train, batch_size = 10, epochs = 100)
```

Output:



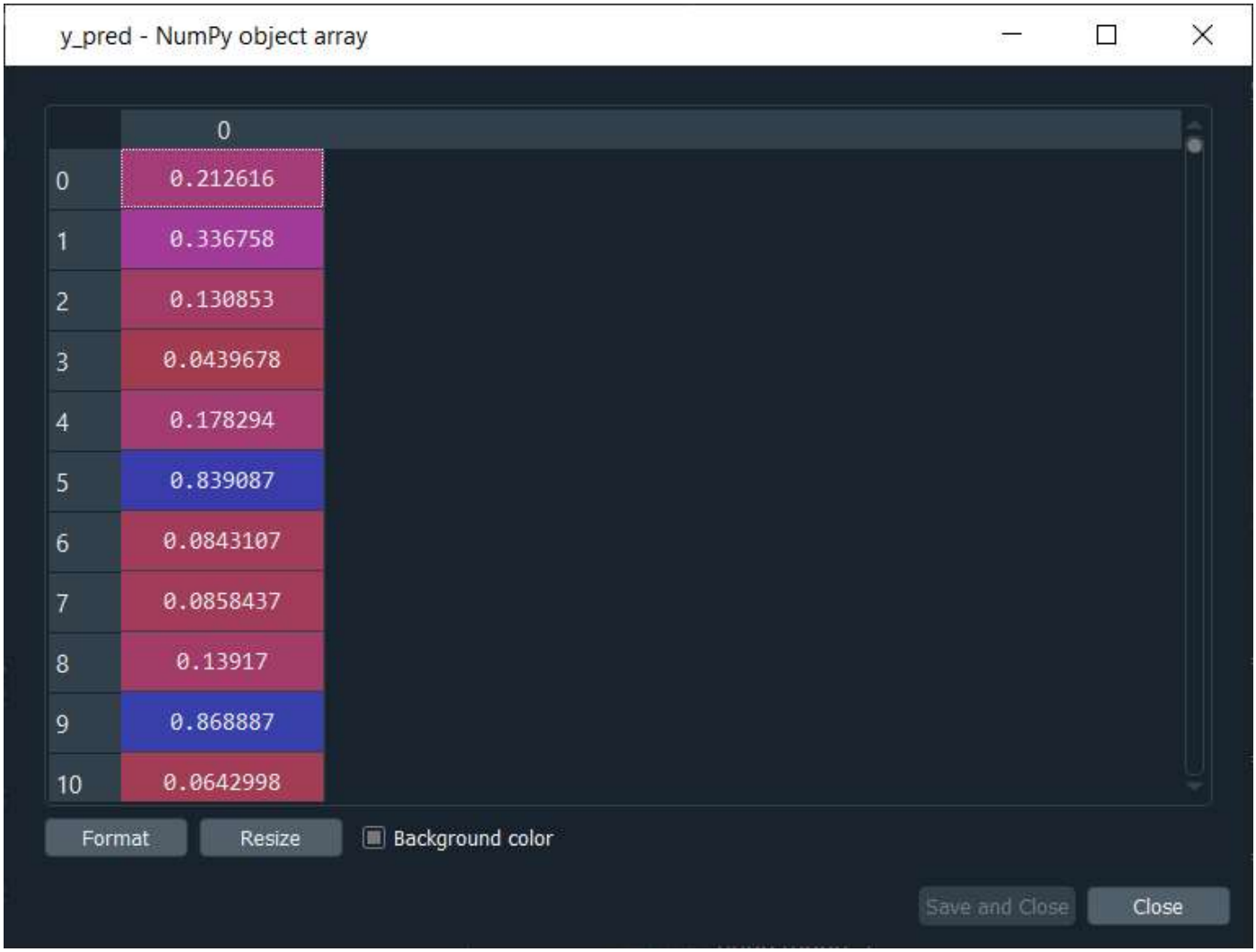
From the output image given above, you can see that our model is ready and has reached an **accuracy** of **84%** approximately, so this how a stochastic gradient descent algorithm is performed.

Part3: Making the Predictions and Evaluating the Model

Since we are done with training the ANN on the training set, now we will make the predictions on the set.

```
y_pred = classifier.predict(X_test)
```

Output:



From the output image given above, we can see all the probabilities that the 2,000 customers of the test set will leave the bank. For example, if we have a look at first probability, i.e., 21% means that this first customer of the test set, indexed by zero, has a 20% chance to leave the bank.

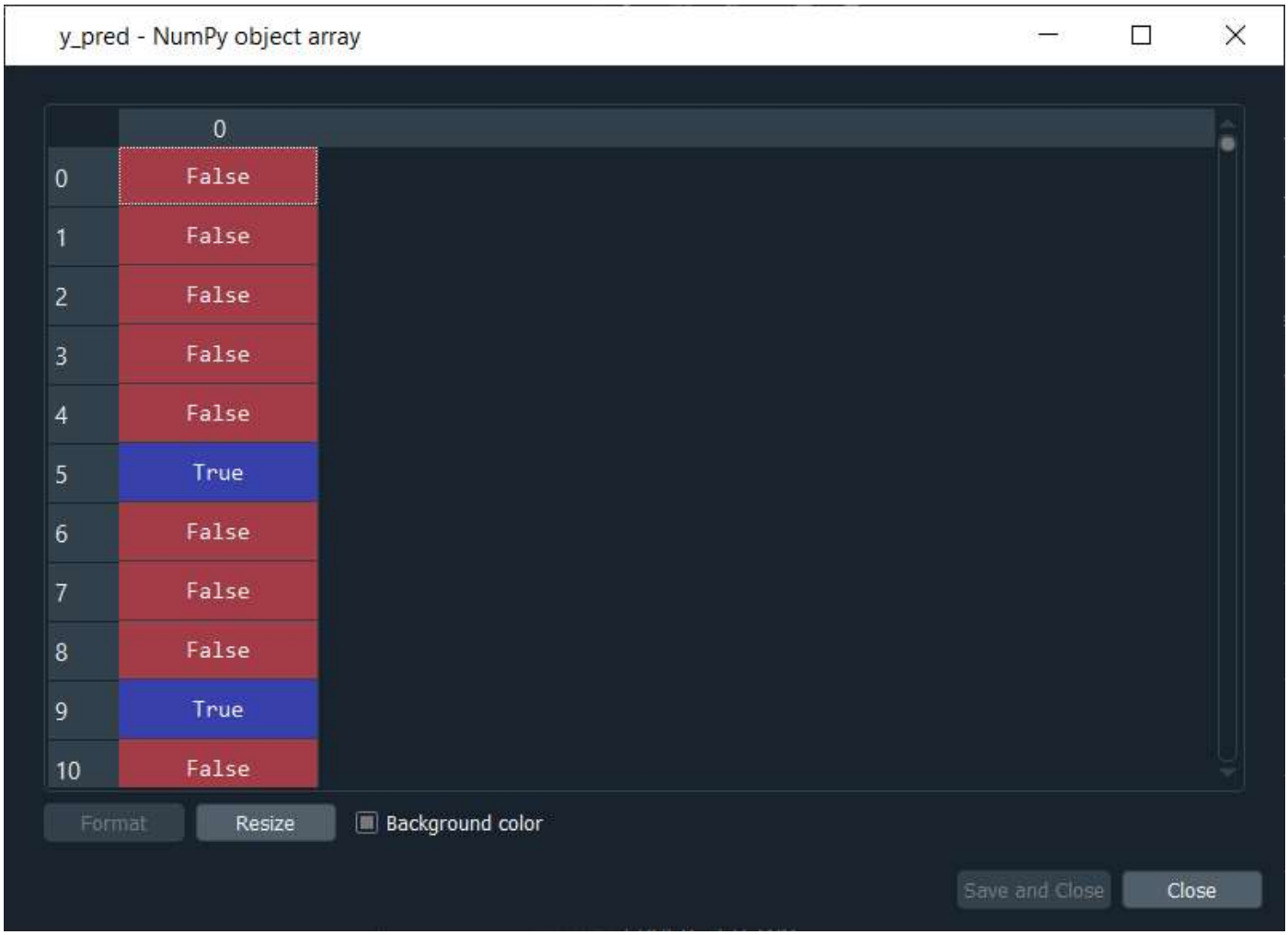
Since the predicted method returns the probability of the customers leave the bank and in order to use this confusion matrix, we don't need these probabilities, but we do need the predicted results in the form of True or False. So, we need to transform these probabilities into the predicted result.

We will choose a threshold value to decide when the predicted result is one, and when it is zero. So, we predict **1** over the threshold and **0** below the threshold as well as the natural threshold that we will take is **0.5**, i.e., 50%. If the **y_pred** is larger, then it will return True else False.

```
y_pred = (y_pred > 0.5)
```

Now, if we have a look at **y_pred**, we will see that it has updated the results in the form of "**False**" or "**True**".

Output:

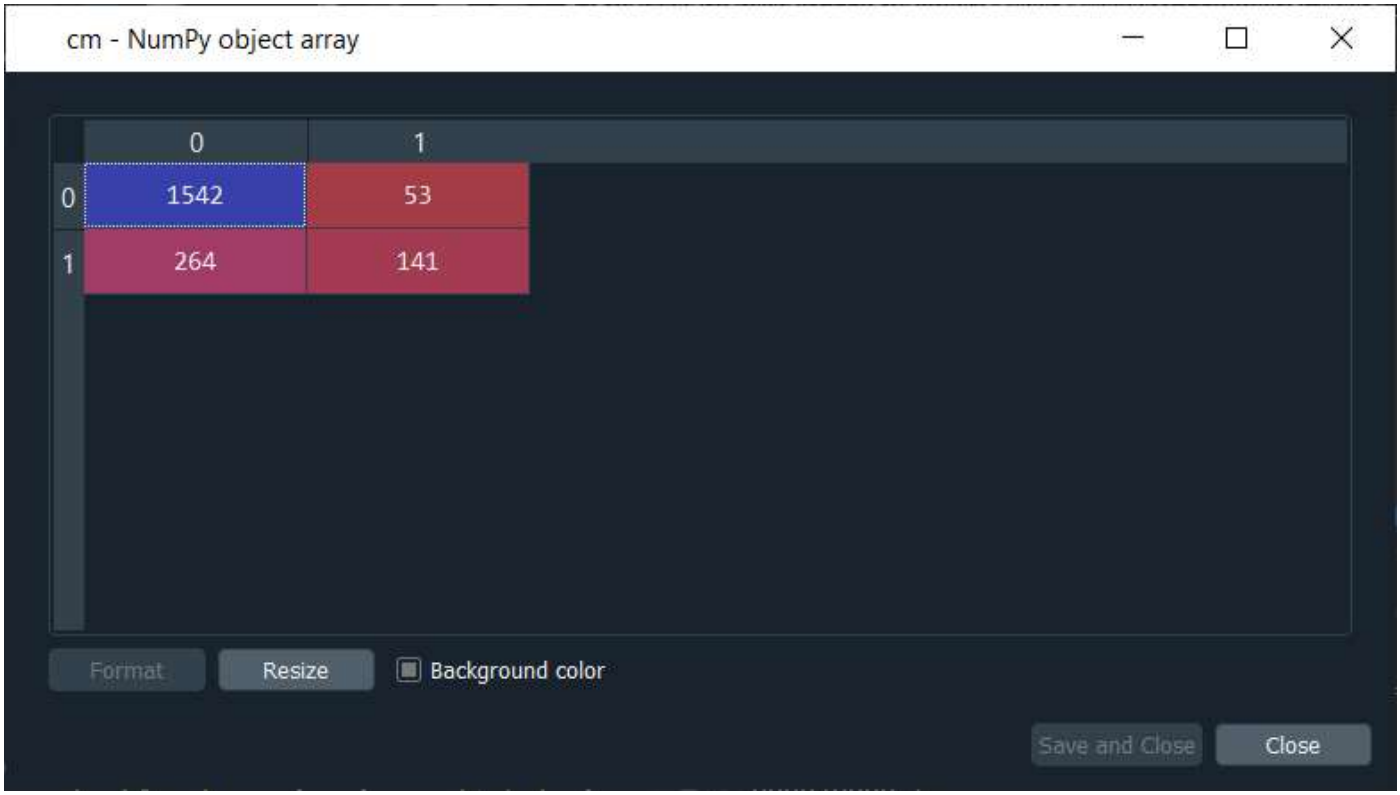


So, the first five customers of the test set don't leave the bank according to the model, whereas the sixth customer in the test set leaves the bank.

Next, we will execute the following code to get the confusion matrix.

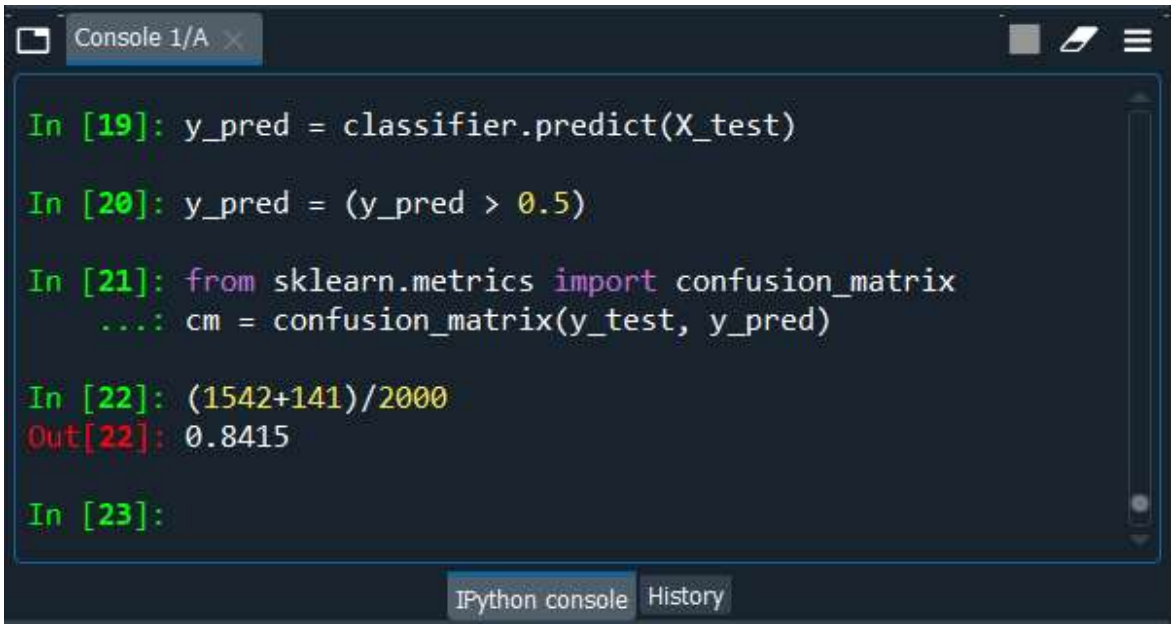
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
```

Output:



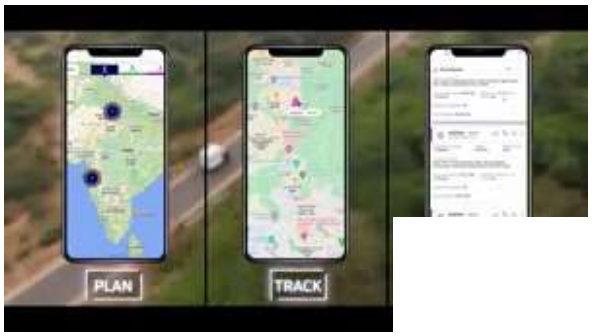
From the output given above, we can see that out of 2000 new observations; we get 1542+141= **1683** correct predictions 264+53= **317** incorrect predictions.

So, now we will compute the accuracy on the console, which is the number of correct predictions divided by the total number of predictions.



So, we can see that we got an accuracy of 84% on new observations on which we didn't train our ANN, even that get a good amount of accuracy. Since this is the same accuracy that we obtained in the training set but obtained here on the test set too.

So, eventually, we can validate our model, and now the bank can use it to make a ranking of their customers, ranked by their probability to leave the bank, from the customer that has the highest probability to leave the bank, down to the customer that has the lowest probability to leave the bank.



← Prev

Next →

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

Learn Latest Tutorials

Splunk tutorial
Splunk

SPSS tutorial
SPSS

Swagger tutorial
Swagger

T-SQL tutorial
Transact-SQL

Tumblr tutorial
Tumblr

React tutorial
ReactJS

Regex tutorial
Regex

Reinforcement learning tutorial
Reinforcement Learning

R Programming tutorial
R Programming

RxJS tutorial
RxJS

React Native tutorial
React Native

Python Design Patterns
Python Design Patterns

Python Pillow tutorial
Python Pillow

Python Turtle tutorial
Python Turtle

Keras tutorial
Keras

Preparation

Aptitude
Aptitude

Logical Reasoning
Reasoning

Verbal Ability
Verbal Ability

Interview Questions
Interview Questions

Company Interview Questions

Company Questions

Trending Technologies

Artificial Intelligence

Artificial Intelligence

AWS Tutorial

AWS

Selenium tutorial

Selenium

Cloud Computing

Cloud Computing

Hadoop tutorial

Hadoop

ReactJS Tutorial

ReactJS

Data Science Tutorial

Data Science

Angular 7 Tutorial

Angular 7

Blockchain Tutorial

Blockchain

Git Tutorial

Git

Machine Learning Tutorial

Machine Learning

DevOps Tutorial

DevOps

B.Tech / MCA

DBMS tutorial

DBMS

Data Structures tutorial

Data Structures

DAA tutorial

DAA

Operating System

Operating System

Computer Network tutorial

Compiler Design tutorial

Computer Organization and Architecture

Discrete Mathematics Tutorial

Computer Network
Compiler Design
Computer Organization
Discrete Mathematics

Ethical Hacking
Ethical Hacking

Computer Graphics Tutorial
Computer Graphics

Software Engineering
Software Engineering

HTML tutorial
Web Technology

Cyber Security tutorial
Cyber Security

Automata Tutorial
Automata

C Language tutorial
C Programming

C++ tutorial
C++

Java tutorial
Java

.Net Framework tutorial
.Net

Python tutorial
Python

List of Programs
Programs

Control Systems tutorial
Control System

Data Mining Tutorial
Data Mining

Data Warehouse Tutorial
Data Warehouse

Perceptron in Machine Learning

In Machine Learning and Artificial Intelligence, Perceptron is the most commonly used term for all folks. It is the primary step to learn Machine Learning and Deep Learning technologies, which consists of a set of weights, input values or scores, and a threshold. ***Perceptron is a building block of an Artificial Neural Network***. Initially, in the mid of 19th century, **Mr. Frank Rosenblatt** invented the Perceptron for performing certain calculations to detect input data capabilities or business intelligence. Perceptron is a linear Machine Learning algorithm used for supervised learning for various binary classifiers. This algorithm enables neurons to learn elements and processes them one by one during preparation. In this tutorial, "Perceptron in Machine Learning," we will discuss in-depth knowledge of Perceptron and its basic functions in brief. Let's start with the basic introduction of Perceptron.

What is the Perceptron model in Machine Learning?

Perceptron is Machine Learning algorithm for supervised learning of various binary classification tasks. Further, ***Perceptron is also understood as an Artificial Neuron or neural network unit that helps to detect certain input data computations in business intelligence***.

Perceptron model is also treated as one of the best and simplest types of Artificial Neural networks. However, it is a supervised learning algorithm of binary classifiers. Hence, we can consider it as a single-layer neural network with four main parameters, i.e., **input values, weights and Bias, net sum, and an**

activation function.

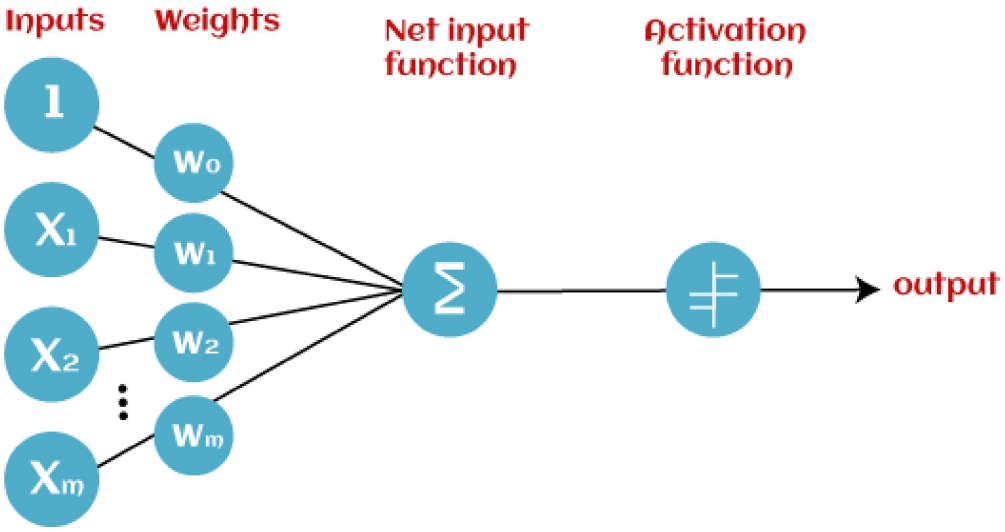
What is Binary classifier in Machine Learning?

In Machine Learning, binary classifiers are defined as the function that helps in deciding whether input data can be represented as vectors of numbers and belongs to some specific class.

Binary classifiers can be considered as linear classifiers. In simple words, we can understand it as a ***classification algorithm that can predict linear predictor function in terms of weight and feature vectors.***

Basic Components of Perceptron

Mr. Frank Rosenblatt invented the perceptron model as a binary classifier which contains three main components. These are as follows:



- **Input Nodes or Input Layer:**

This is the primary component of Perceptron which accepts the initial data into the system for further processing. Each input node contains a real numerical value.

- **Wight and Bias:**

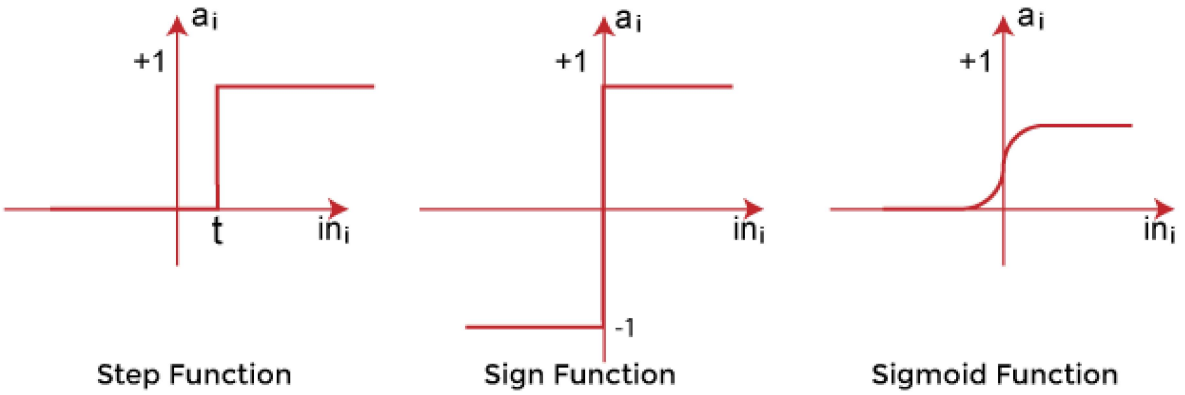
Weight parameter represents the strength of the connection between units. This is another most important parameter of Perceptron components. Weight is directly proportional to the strength of the associated input neuron in deciding the output. Further, Bias can be considered as the line of intercept in a linear equation.

- **Activation Function:**

These are the final and important components that help to determine whether the neuron will fire or not. Activation Function can be considered primarily as a step function.

Types of Activation functions:

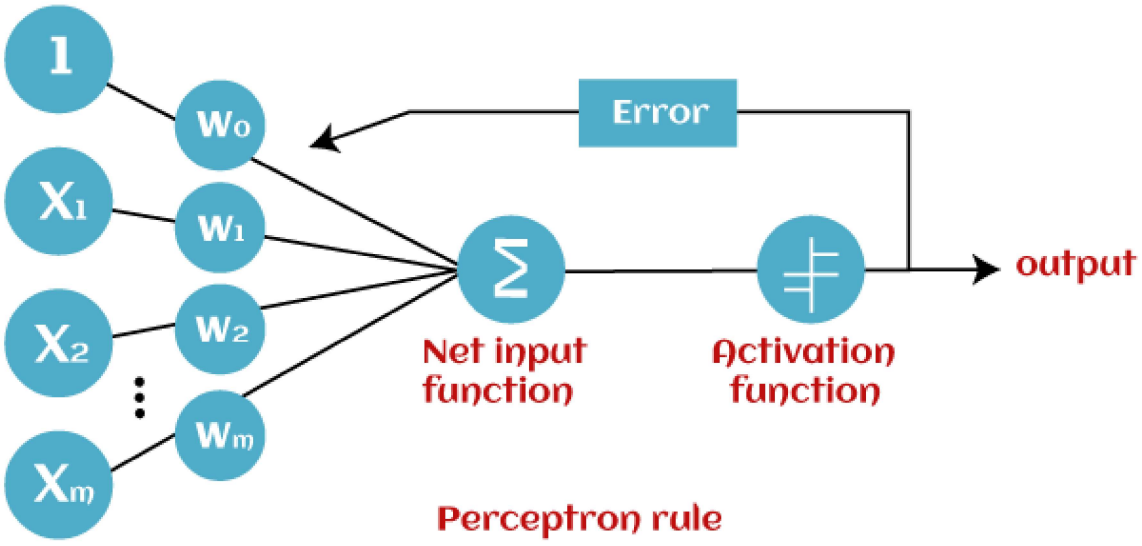
- Sign function
- Step function, and
- Sigmoid function



The data scientist uses the activation function to take a subjective decision based on various problem statements and forms the desired outputs. Activation function may differ (e.g., Sign, Step, and Sigmoid) in perceptron models by checking whether the learning process is slow or has vanishing or exploding gradients.

How does Perceptron work?

In Machine Learning, Perceptron is considered as a single-layer neural network that consists of four main parameters named input values (Input nodes), weights and Bias, net sum, and an activation function. The perceptron model begins with the multiplication of all input values and their weights, then adds these values together to create the weighted sum. Then this weighted sum is applied to the activation function 'f' to obtain the desired output. This activation function is also known as the **step function** and is represented by **'f'**.



This step function or Activation function plays a vital role in ensuring that output is mapped between required values (0,1) or (-1,1). It is important to note that the weight of input is indicative of the strength of a node. Similarly, an input's bias value gives the ability to shift the activation function curve up or down.

Perceptron model works in two important steps as follows:

Step-1

In the first step first, multiply all input values with corresponding weight values and then add them to determine the weighted sum. Mathematically, we can calculate the weighted sum as follows:

∑wi*xi = x1*w1 + x2*w2 +...wn*xn

Add a special term called **bias 'b'** to this weighted sum to improve the model's performance.

∑wi*xi + b

Step-2

In the second step, an activation function is applied with the above-mentioned weighted sum, which gives us output either in binary form or a continuous value as follows:

Y = f(∑wi*xi + b)

Types of Perceptron Models

Based on the layers, Perceptron models are divided into two types. These are as follows:

1. Single-layer Perceptron Model
2. Multi-layer Perceptron model

Single Layer Perceptron Model:

This is one of the easiest Artificial neural networks (ANN) types. A single-layered perceptron model consists feed-forward network and also includes a threshold transfer function inside the model. The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes.

In a single layer perceptron model, its algorithms do not contain recorded data, so it begins with inconstantly allocated input for weight parameters. Further, it sums up all inputs (weight). After adding all inputs, if the total sum of all inputs is more than a pre-determined value, the model gets activated and shows the output value as +1.

If the outcome is same as pre-determined or threshold value, then the performance of this model is stated as satisfied, and weight demand does not change. However, this model consists of a few discrepancies triggered when multiple weight inputs values are fed into the model. Hence, to find desired output and minimize errors, some changes should be necessary for the weights input.

"*Single-layer perceptron can learn only linearly separable patterns.*"

Multi-Layered Perceptron Model:

Like a single-layer perceptron model, a multi-layer perceptron model also has the same model structure but has a greater number of hidden layers.

The multi-layer perceptron model is also known as the Backpropagation algorithm, which executes in two stages as follows:

- **Forward Stage:** Activation functions start from the input layer in the forward stage and terminate on the output layer.
- **Backward Stage:** In the backward stage, weight and bias values are modified as per the model's requirement. In this stage, the error between actual output and demanded originated backward on the output layer and ended on the input layer.

Hence, a multi-layered perceptron model has considered as multiple artificial neural networks having various layers in which activation function does not remain linear, similar to a single layer perceptron model. Instead of linear, activation function can be executed as sigmoid, TanH, ReLU, etc., for deployment.

A multi-layer perceptron model has greater processing power and can process linear and non-linear patterns. Further, it can also implement logic gates such as AND, OR, XOR, NAND, NOT, XNOR, NOR.

Advantages of Multi-Layer Perceptron:

- A multi-layered perceptron model can be used to solve complex non-linear problems.
- It works well with both small and large input data.
- It helps us to obtain quick predictions after the training.
- It helps to obtain the same accuracy ratio with large as well as small data.

Disadvantages of Multi-Layer Perceptron:

- In Multi-layer perceptron, computations are difficult and time-consuming.
- In multi-layer Perceptron, it is difficult to predict how much the dependent variable affects each independent variable.
- The model functioning depends on the quality of the training.

Perceptron Function

Perceptron function ''f(x)'' can be achieved as output by multiplying the input 'x' with the learned weight coefficient 'w'.

Mathematically, we can express it as follows:

f(x)=1; if w.x+b>0

otherwise, f(x)=0

- 'w' represents real-valued weights vector
- 'b' represents the bias
- 'x' represents a vector of input x values.

Characteristics of Perceptron

The perceptron model has the following characteristics.

1. Perceptron is a machine learning algorithm for supervised learning of binary classifiers.
2. In Perceptron, the weight coefficient is automatically learned.
3. Initially, weights are multiplied with input features, and the decision is made whether the neuron is fired or not.
4. The activation function applies a step rule to check whether the weight function is greater than zero.
5. The linear decision boundary is drawn, enabling the distinction between the two linearly separable classes +1 and -1.
6. If the added sum of all input values is more than the threshold value, it must have an output signal; otherwise, no output will be shown.

Limitations of Perceptron Model

A perceptron model has limitations as follows:

- The output of a perceptron can only be a binary number (0 or 1) due to the hard limit transfer function.
- Perceptron can only be used to classify the linearly separable sets of input vectors. If input vectors are non-linear, it is not easy to classify them properly.

Future of Perceptron

The future of the Perceptron model is much bright and significant as it helps to interpret data by building intuitive patterns and applying them in the future. Machine learning is a rapidly growing technology of Artificial Intelligence that is continuously evolving and in the developing phase; hence the future of perceptron technology will continue to support and facilitate analytical behavior in machines that will, in turn, add to the efficiency of computers.

The perceptron model is continuously becoming more advanced and working efficiently on complex problems with the help of artificial neurons.

Conclusion:

In this article, you have learned how Perceptron models are the simplest type of artificial neural network which carries input and their weights, the sum of all weighted input, and an activation function. Perceptron models are continuously contributing to Artificial Intelligence and Machine Learning, and these models are becoming more advanced. Perceptron enables the computer to work more efficiently on complex problems using various Machine Learning technologies. The Perceptrons are the fundamentals of artificial neural networks, and everyone should have in-depth knowledge of perceptron models to study deep neural networks.

← Prev Next →

Python Design Patterns

Python Pillow

Python Turtle

Keras

Preparation

Aptitude

Reasoning

Verbal Ability

Interview Questions

Company Questions

Trending Technologies

Artificial Intelligence

AWS

Selenium

Cloud Computing

Hadoop

ReactJS

Data Science

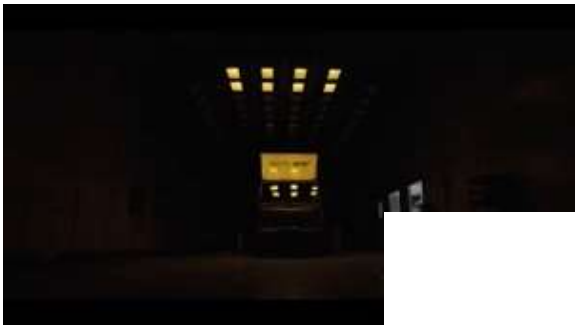
Angular 7

Blockchain

Git

Machine Learning

DevOps



B.Tech / MCA

DBMS tutorial DBMS	Data Structures tutorial Data Structures	DAA tutorial DAA	Operating System Operating System
Computer Network tutorial Computer Network	Compiler Design tutorial Compiler Design	Computer Organization and Architecture Computer Organization	Discrete Mathematics Tutorial Discrete Mathematics
Ethical Hacking Ethical Hacking	Computer Graphics Tutorial Computer Graphics	Software Engineering Software Engineering	html tutorial Web Technology
Cyber Security tutorial Cyber Security	Automata Tutorial Automata	C Language tutorial C Programming	C++ tutorial C++
Java tutorial Java	.Net Framework tutorial .Net	Python tutorial Python	List of Programs Programs

Control Systems tutorial

Control System

Data Mining Tutorial

Data Mining

Data Warehouse Tutorial

Data Warehouse