ਜਗਤ ਗੁਰੂ ਨਾਨਕ ਦੇਵ
ਪੰਜਾਬ ਸਟੇਟ ਓਪਨ ਯੂਨੀਵਰਸਿਟੀ
ਪਟਿਆਲਾ

JAGAT GURU NANAK DEV
PUNJAB STATE OPEN UNIVERSITY, PATIALA

(Established by Act No. 19 of 2019 of the Legislature of State of Punjab)

The Motto of the University
(SEWA)

SKILL ENHANCEMENT **EMPLOYABILITY** **WISDOM**
ACCESSIBILITY



DIPLOMA IN AI & DATA SCIENCE
SEMESTER-II
Course: Machine Learning Lab
Course Code: DAIDS-2-01P

ADDRESS: C/28, THE LOWER MALL, PATIALA-147001
WEBSITE: www.psou.ac.in

SELF-INSTRUCTIONAL STUDY MATERIAL FOR JGND PSOU, ALL COPYRIGHTS WITH JGND PSOU, PATIALA

DAIDS-2-01P: Machine Learning Lab

Total Marks: 50
External Marks: 35
Internal Marks: 15
Credits: 2
Pass Percentage: 40%

Course: Machine Learning Lab	
Course Code: DAIDS-2-01P	
Course Outcomes (COs) After the completion of this course, the students will be able to:	
CO1	Apply a perceptron to solve binary classification problems.
CO2	Apply ADALINE and MADALINE to solve binary classification problems.
CO3	Write code to implement the backpropagation algorithm from scratch.
CO4	Implement and experiment with different clustering algorithms.
CO5	Work with real-world datasets to apply machine learning algorithms or training neural networks.

Detailed List of Programmes:

Programme No.	**Name of Programme**
P1	Extract the data from database using python
P2	Implementation of Linear Regression
P3	Implementation of Logistic regression
P4	Implementation of Decision tree classification
P5	Implementation of K-nearest Neighbor
P6	Implement the Perceptron Learning rule works for OR Gate training.
P7	Implement the ADALINE works for AND Gate training.
P8	Implement the MADALINE works for XOR Gate training
P9	Build Artificial Neural Network model with back propagation
P10	Implementing K-means Clustering
P11	Implementation of Unsupervised Learning Algorithm ART1
P12	Implementation of Unsupervised Learning Algorithm ART2

MACHINE LEARNING

UNIT I: INTRODUCTION TO MACHINE LEARNING

STRUCTURE

1.0 Objectives

1.1 Introduction

1.2 What is Machine Learning?

1.3 Categories of Machine learning

1.4 Applications of Machine learning

1.5 Machine learning as a future

1.6 Data pre-processing

1.7 Why data pre-processing

1.8 Steps in data-processing

1.9 Self-check questions

1.10 Summary

1.11 Unit end questions

1.0 OBEJCTIVES

- Learn the fundamentals of machine learning and its categorization.
- Understand the applications where machine learning is used.
- Get deep insights about how to do data pre-processing on the machine learning datasets.

1.1 INTRODUCTION

This module targets understanding the concept of machine learning along with its categorization. This module also discusses applications of machine learning, its future, and a detailed discussion on data pre-processing steps along with python program examples. This module targets graduate students who want to learn machine learning using python programming. Machine learning is a sub-field of artificial intelligence and is currently used for sentiment analysis, video surveillance, diseases detection, and recognition, etc. The main target of this module is to inculcate the basics of machine learning to the learners so that they can do data pre-processing on the datasets before moving with higher steps of machine earning.

1.2 WHAT IS MACHINE LEARNING?

It is a sub-field of artificial intelligence (AI) that encompasses mathematical computational models to predict the outcomes by understanding data. Machine learning algorithms are firstly trained on the historical data such that they can predict the output values for the newly observed data. The question that arises is that machine learning is similar to human brain learning. As the name suggests, machine learning makes computer learning similar to humans based on their „ability to learn". Machine learning is popularly for crop diseases prediction, spam filtering, business process automation, and emotion recognition, etc.

1.3 CATEGORIES OF MACHINE LEARNING

How does a machine learning algorithm learn for accurately predicting the new outcomes help in the categorization of machine learning algorithms? These are categorized into four types and these are mentioned below:

a) Supervised learning
b) Unsupervised learning
c) Semi-supervised learning
d) Reinforcement learning

a) Supervised Learning

In this learning, the objective is to build a general rule that helps in making a relationship between the inputs given to the algorithm with the outputs obtained after executing the algorithm. The algorithms are trained with input and output data till a certain level of accuracy is achieved. Now, testing is done on new data to check its predicting accuracy. You can say that both input and output are specified, the algorithm targets to learn by mapping input to the output. Supervised learning helps in performing the tasks like classification and regression.

Classification: It helps in predicting the class label for a given input data. For example, the algorithm is trained with many images along with its corresponding class labels. The testing accuracy will be high when you provide new images to the algorithm and it can classify them correctly according to their class labels. Here, the class labels are discrete categories.

Regression: The class labels in regression are continuous instead of discrete categories. Example: stock market prediction, here the algorithms are trained based on historical data and in future predict the new stock price. In input, you have one or more predictor variables that help in predicting continuous outcomes.

b) Unsupervised Learning

This learning figures out the hidden patterns from the unlabeled data. The training is performed on the unlabeled data and the algorithm makes a meaningful connection by observing hidden patterns and when any unlabeled data is inputted to the algorithm while testing, the algorithm uses the knowledge of training while predicting the outcome. It performs many tasks like clustering, high dimension visualization, anomaly detection, and dimensionality reduction [1].

Clustering: By checking the similarity, spilt the similar data into clusters.

High Dimension Visualization: This helps in visualizing the data that is having high dimensions.

Anomaly Detection: Figure out the unusual data points in the data.

Dimensionality Reduction: Minimise the number of variables in the data such that it does not affect the model training and testing.

c) Semi-supervised Learning

In this type of learning, you have a large amount of data and some of the data is labelled unlike unsupervised learning where data is unlabelled and in supervised learning, data is labelled. The algorithm‟s performance enhances when the whole data is labelled but data labelling is an expensive and time-consuming process. This semi-supervised learning lies between supervised and unsupervised learning. Some of the tasks that are performed by this type of learning are machine translation, fraud detection, etc [2].

Machine translation: Translate a language when a full dictionary of words is not available.

Fraud detection: Finding out the frauds when you have only some positive examples

d) Reinforcement learning:

An algorithm works for a specific goal with a certain set of rules for achieving the specific goal. There are rewards and punishments while reaching your specific goal. Positive rewards are given when your programmed algorithm is reaching close to your goal and punishments are given when it is going farther from your goal. This learning can perform certain tasks like driverless cars, video gameplay, etc [1].

Driverless cars: This algorithm helps the car by learning from the dynamic environment and make them self-dependent.

Video gameplay: Bots are being trained using reinforcement learning to play a vast number of video games.

1.4 APPLICATIONS OF MACHINE LEARNING

In the current era, machine learning is the buzz word and it is used for daily routine tasks as well as for complex tasks. Some of the machine learning applications [3] are mentioned below:

a) Product Recommendations:

Big Giant e-commerce companies use product recommendations for their e-commerce website customers. They use machine learning and AI algorithms on the backend for product recommendations by tracking your past purchases and searching patterns.

b) Image Recognition:

This is one of the important applications of machine learning. Detect an object from a digital image is the image recognition aspect. Object detection like face recognition, eyes recognition, patterns recognition from digital images. This technique is further used for emotion recognition and fraud analysis etc.

c) Sentiment Analysis:

This is one of the real-time applications that helps in knowing the emotion or opinion state of mind of speakers or writers. A sentiment analyzer will analyze your written text, face, or voice for predicting your state of mind whether you are happy, sad, angry, nervous, etc. These analyses are used for decision-making applications and review-based websites.

d) Prediction of potential heart failure

Machine learning algorithms help in finding out the patterns in the patient‟s cardiovascular history, this helps the doctors to do diagnoses without working on the patient‟s health records. These algorithms minimize the chances of the wrong diagnosis along with saves time for analyzing available history information.

e) Video Surveillance

These are the most progressive applications of machine learning and AI. It provides better prospects to get useful information from automated surveillance devices rather than any other source. Machines can look for the objects by 24*7 as compared to human brains. Video surveillance is used for traffic monitoring and management, parking lots, theft prevention, hospital operation monitoring, abnormal event detections, etc.

f) Chatbots

Nowadays, chatbots are used to answer customer queries in different applications like banking, purchasing, stock market, etc. These applications have the option “chat with us”. Chatbots use the concept of machine learning (decision tree mostly) while resolving different

customer queries. Machine learning concept helps the machines to learn quickly, fast reply and satisfy customers.

g) Virtual Personal Assistants (VPAs)

VPAs help to discover information and set instructions when you ask them something over voice. Discover information like “what are the flights from Mumbai to Delhi” and set instruction commands like “set alarm for 5“clock in the morning. VPAs can handle your queries and instructions as they are trained and tested using machine learning concepts. Some of the popular VPAs are Alexa, Google, Siri, etc. They are also integrated with different platforms like Smart Speakers (Google Home, Amazon Echo), Mobile Apps (Google Allo), and Smartphones (Samsung Bixby) [4], etc.

h) Spam and Malware Filtering

Spam filtering is an approach that is used by email clients. Machine learning is backing up the spam filters for regular and continuous updating. When rules are made to do the spam filtering, they sometimes fail to do as spammers adopt the latest tricks to befool them. But, when machine learning-backed spam filters like Decision tree filters, multi-layer perceptron, they are toughed to be fooled by spammers. Lakhs of malware are detected daily and machine learning algorithms help in detecting and figuring out the coding patterns in this malware and protect the system from further damage.

i) Results Refinement for Search Engine

Yahoo, Bing, Google, and other search engines use machine learning algorithms in the backyard to improve the search results of your search string. One of the ways for learning for the search engine is that when you put a search string in the search engine and the results which came out after executing the string shows that you open the top results of the first page, the search engine assumes that results are according to your query. If you go to the 3rd or 4th page of search results, the search engine assumes that results are not according to the search string. This is one of the ways to improve the search results.

j) Email Spam and Malware filtering

You can see when you receive a mail, it is filtered automatically as spam, normal or important. The important emails are received in the inbox with important symbols whereas spam emails are in the spam box. This whole process is done with the help of machine learning. Gmail uses some of the spam filters and these are:

- Rule-based filters
- Content filters
- Permission filters
- Header filters

The machine learning algorithms that are used for malware detection and email spam filtering are Decision Tree, Multi-layer perceptron, etc.

1.5 MACHINE LEARNING AS A FUTURE

The computer algorithms that are based on machine learning learn reflexively from the dynamic environment through their experiences and many trials. These algorithms are capable of predicting output without human intervention and with utmost precision. Nowadays as well as in the future, machine learning will remain a buzzword. At present, these types of algorithms are used in every common domain like digital marketing, banking, health care, insurance sector, stock market, etc. The development of machine learning algorithms will impact and change the lives of the common man that was impossible 10 years back. The new machine learning algorithms will be capable to learn through trials and experiences, reflex, adjust and self-sufficiently act in the dynamic environment as compared with predefined rules for innovation methods [5]. In the future, the following are the areas where you can find futuristic advancements in machine learning algorithms.

a) Utmost accurate results for Web Search Engines
b) Improved Tailor-made applications customization accuracy
c) Rise of automated Robots or self-learning systems
d) A sudden and great increase in the usage of machine learning algorithms in quantum computing
e) More accurate predictions in the healthcare sector for disease prediction and drug development processes.
f) Automated end to end model development process
g) Machine learning will be the new era for manufacturing processes in the coming ten years.

1.6 DATA PRE-PROCESSING

One of the important steps for enhancing the quality of data in machine learning. To extract meaningful information from the data, this step plays an important role. This step cleans and organizes the raw data in such a way so that this data is useful for training the machine learning models. In other words, it can be said that raw data is transferred into a readable and understandable format.

1.7 WHY DATA PRE-PROCESSING

As the real-world data is unreliable, inadequate, inaccurate, erroneous and it is not suitable for training the models. This scenario creates the need for data pre-processing. In this initial step of the machine learning process, the raw data is cleaned, completed, formatted, and organized such that training for models can be performed using this data [6].

1.8 STEPS IN DATA PRE-PROCESSING

There are seven steps while working on the data pre-processing [6] and these are listed as with explanation.

1) Get the dataset

This is the initial step of data preprocessing in machine learning. To run your machine learning models, you must have a pertinent dataset. To get a proper dataset, data should be collected from many different primary or secondary sources and should be properly formatted accordingly to the use cases. For example, the textile industry dataset should contain information about textile data, citrus fruit disease dataset should contain information about

different types of citrus fruits and their disease data. Many online sources are also available to download the data like Kaggle, UCI machine learning repository, etc. The private dataset should be made by doing surveys, interviews and using different Python APIs.

2) Import all relevant libraries

This is the second step of data pre-processing. Python programming language is one of the most popular and leading languages to solve the complex problems and tasks associated with machine learning and data analytics. To develop machine learning programs for particular tasks, python libraries are extensively used and beneficial. These predefined libraries can do data-specific pre-processing tasks. Some of the popular libraries are:

a) NumPy: This is one of the python cores packages for doing scientific and mathematical operations. It stands for Numerical Python and was formed by Travis Oliphant in the year 2005. This package consists of functions that can do linear algebra, arrays, and Fourier transforms operations.
b) Pandas: This open-source library was built by McKinney in 2008 and used to perform data analysis, data cleaning, data manipulation, and data exploration. This library is mainly used to import the different datasets.
c) Matplotlib: This open-source library was formed by John D. Hunter. This is used to plot quality charts and figures. This is popularly known as a 2D plotting library. Most segments of this library are written in Python and other segments are written in Javascript, C, and Objective-C.

3) Import the dataset

Before running the machine learning model, it is important to import the dataset. You can import the dataset in many ways, one of the ways is to set the current directory as a working set and another way is to give the absolute path of the file. After setting up the path, use read_csv() of the pandas' library to import the dataset in the program, this function can read a CSV file either it is stored locally on a computer or with the help of a URL.

Example: data = pd.read_csv(r'E:\Jagat Open State University\Machine learning\job1.csv')

where job1.csv is the dataset having attributes like country, age, expected salary and job granted respectively. Here, country, age and ExpectedSalary are independent variables whereas JobGranted is a dependent variable. Extraction of these variables is an important step before running any machine learning model. For extraction of these variables from the dataset, pandas library function “iloc[ ]” will be used. It helps in the extraction of selected rows and columns.

x= data.iloc[:,:-1].values

Here, in the above line code of iloc[ ] function, first colon represents all rows and second colon represents all columns. The value of -1 indicates that leave the last column from the dataset as it is an dependent variable. After executing the above code line, the result will be of independent variables and the result is like:

[['India' 28.0 45000.0]

['USA' 28.0 34000.0]
['Spain' 36.0 42000.0]
['USA' 45.0 90000.0]
['Spain' 38.0 nan]
[nan 54.0 89000.0]
['India' nan 67000.0]
['Spain' 53.0 88000.0]
['Spain' 32.0 40000.0]
['USA' 44.0 98000.0]]

y= data.iloc[:,3].values

When you run the above code line, the first colon represents all rows and 3 represents only 3rd column. In other words, it can be said that only the last column (i.e., JobGranted) with all rows is extracted. This is the dependent variable. The output of the above line code is as:

array(['No', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'No'],dtype=object)

4) Identify and Handle the missing values in the dataset

When you are dealing with the real-time dataset you will find missing values for many attributes. This can be due to incomplete extraction, failure to load the complete information, or corrupt data. One of the major challenges faced by software developers is to handle the missing values in such a way that robust models will be generated. There are many ways of handling missing values and for these python libraries like pandas, NumPy and Scikit are mostly used.

Use the below-mentioned code line to know which attributes in the dataset have missing values:
data.isnull().sum()

The output is:

Country 0
Age 1
ExpectedSalary 1
JobGranted 0
dtype: int64

From above, it can be seen that age and ExpectedSalary have one missing value each.

To handle these missing values, different ways are used and some of the ways are mentioned below:

i. Deleting rows having missing values

One of the most commonly used methods to handle missing values. Delete the particular row if it has a missing or null value and deletes the particular column if it has more than 50% of missing or null values. This scenario is only suitable if a large dataset is available. Important things that have to be kept in mind while deleting data is that no biasness should be added and there is always a loss of information which may lead to inaccurate predictions [7].

In the above dataset, you have seen two rows are having missing values and these can be dropped and checked whether these rows have been dropped or not using the below lines of code:

```
data.dropna(inplace=True)  # Drop the rows

data.isnull().sum()        # Check missing rows
```

ii. Replacing with Mean/Median/Mode

This approximation statistical method is applied to the attributes/features that are having numerical data like age or ExpectedSalary. Here, calculate the mean, median, or mode of the feature and replace the missing value with the calculated value. This method adds variance to the dataset but is better than removing rows and columns from the dataset [7]. The code lines are:

a) Replacing with mean value

```
data['Age'].isnull().sum()  # Output is 1

data['Age'].mean()          # Output is 39.77777777777778

data['Age'].replace(np.NaN,data['Age'].mean())

#Output

0  28.000000
1  28.000000
2  36.000000
3  45.000000
4  38.000000
5  54.000000
6  39.444444        #Bold value replaced with mean
7  53.000000
8  32.000000
9  44.000000
Name: Age, dtype: float64
```

b) Replacing with median value

```
data['Age'].isnull().sum()  # Output is 1

data['Age'].median()        # Output is 38.0
```

```
data['Age'].replace(np.NaN,data['Age'].median())

#Output

0   25.0
1   28.0
2   36.0
3   45.0
4   38.0
5   54.0
6   38.0   #Bold value replaced with median
7   53.0
8   32.0
9   44.0
Name: Age, dtype: float64
```

c) Replacing with mode value

```
data['Age'].isnull().sum()     # Output is 1

data['Age'].median()              # Output is 28.0

data['Age'].fillna(data['Age'].mode()[0])

#Output

0   28.0
1   28.0
2   36.0
3   45.0
4   38.0
5   54.0
6   28.0   # Bold value replaced with mode value
7   53.0
8   32.0
9   44.0
Name: Age, dtype: float64
```

d) Unique Category Assignment

A categorical feature consists of many possibilities of classes and their missing class values can be assigned with another class like Unknown denoted as „U". This tactic will add more information in the data that changes the variance of the data. Since categorical data must be converted into numerical data using a one-hot encoding such that machine learning algorithms must understand the data.

```
data['Country'].head(7)
```

#Output

0 India
1 USA
2 Spain
3 USA
4 Spain
5 NaN
6 India
Name: Country, dtype: object

The following implementation line will tell that how to put Unknown value „U“ for category value.

data['Country'].fillna('U').head(7)

#Output

0 India
1 USA
2 Spain
3 USA
4 Spain
5 U
6 India
Name: Country, dtype: object

e) Use the machine learning algorithms that support missing values

The algorithms like Random Forest and KNN of machine learning algorithm supports missing values if they are present in the data. These algorithms maintain the high variance that is present in the data.

5) Encode the categorical variable

The dataset job1.csv has two categorical variables and these are Country and JobGranted. As machine learning algorithms creates many problems when they work on the categorical data, so it must be converted into numerical data. For example, country column is converted into numerical data using LabelEncoder() class. This is present in the sci-kit library.

Following lines of code, encode the country 0 for India, 1 for Spain, 2 for USA and 3 for U

from sklearn.preprocessing import LabelEncoder

label_encoder_x= LabelEncoder()

x[:, 0]= label_encoder_x.fit_transform(x[:, 0])

print(x)

#Output

```
[[0 28.0 45000.0]
 [2 28.0 34000.0]
 [1 36.0 42000.0]
 [2 45.0 90000.0]
 [1 38.0 nan]
 [3 54.0 89000.0]
 [0 nan 67000.0]
 [1 53.0 88000.0]
 [1 32.0 40000.0]
 [2 44.0 98000.0]]
```

This encoding of categorical may affect the outcome as algorithms treat these categorical numerical data as some relationship. To avoid this effect, a dummy variable concept is introduced. For the above categories values of 0,1,2 and 3, four columns will be generated having values 0 and 1. Dummy encoding is used for the generation of columns using OneHotEncoder and the code is:

Step 1:

```
dummies = pd.get_dummies(data['Country'])

print(dummies)

#Output
```

India	Spain	USA	
0	1	0	0
1	0	0	1
2	0	1	0
3	0	0	1
4	0	1	0
5	0	0	0
6	1	0	0
7	0	1	0
8	0	1	0
9	0	0	1

Step 2:

```
merg = pd.concat([data,dummies],axis='columns')

print(merg)

#Output
```

Country	Age	ExpectedSalary	JobGranted	India	Spain	USA
0	India	28.0	45000.0	No	1	0	0
1	USA	28.0	34000.0	Yes	0	0	1
2	Spain	36.0	42000.0	Yes	0	1	0
3	USA	45.0	90000.0	No	0	0	1
4	Spain	38.0	NaN	Yes	0	1	0
5	NaN	54.0	89000.0	Yes	0	0	0
6	India	NaN	67000.0	No	1	0	0
7	Spain	53.0	88000.0	Yes	0	1	0
8	Spain	32.0	40000.0	Yes	0	1	0
9	USA	44.0	98000.0	No	0	0	1

Step3:

```
final = merg.drop(['Country','USA'],axis='columns')
print(final)
```

#Output	Age	ExpectedSalary	JobGranted	India	Spain
0	28.0	45000.0	No	1	0
1	28.0	34000.0	Yes	0	0
2	36.0	42000.0	Yes	0	1
3	45.0	90000.0	No	0	0
4	38.0	NaN	Yes	0	1
5	54.0	89000.0	Yes	0	0
6	NaN	67000.0	No	1	0
7	53.0	88000.0	Yes	0	1
8	32.0	40000.0	Yes	0	1
9	44.0	98000.0	No	0	0

6) Dataset Division or Split

The next step in the data pre-processing is the division of the dataset into a training and testing set. The training set is the subset of the dataset and is used to train the machine learning model. Testing set is used for testing the developed model to do the predictions. The dataset is divided generally into the ratio of 70:30 or 80:20. It means 70% or 80% of data is

used for training the model whereas the rest 30% or 20% is used for testing the machine learning model. This division process of data depends upon the shape and size of the dataset.

Firstly look the independent variables after performing the above mentioned steps:

```
x= final.iloc[:,[0,1,3,4,5]].head(10) #Indepenedent Variables

print(x)

#Output
```

Age	ExpectedSalary	India	Spain	U	
0	28.000000	45000.0	1	0	0
1	28.000000	34000.0	0	0	0
2	36.000000	42000.0	0	1	0
3	45.000000	90000.0	0	0	0
4	38.000000	67000.0	0	1	0
5	54.000000	89000.0	0	0	1
6	39.777778	67000.0	1	0	0
7	53.000000	88000.0	0	1	0
8	32.000000	40000.0	0	1	0
9	44.000000	98000.0	0	0	0

Now, look at the data of the dependent variable after performing above mentioned preprocessing steps:

```
y= final.iloc[:,2].head(10) #Indepenedent Variables

print(y)

#Output

0     No
1    Yes
2    Yes
3     No
4    Yes
5    Yes
6     No
7    Yes
8    Yes
9     No
Name: JobGranted, dtype: object
```

Now, divide the dataset into training and testing dataset using the following lines of code:

```
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)

print(x_train)
```

The first line divides the dataset into training and testing randomly. The second line of code contains four variables and these are:

- x_train – features for the training data
- x_test – features for the test data
- y_train – dependent variables for training data
- y_test – independent variable for testing data

The function train_test_split() is used and have four parameters. The first two parameters represent arrays of data and the third parameter denotes the test size and it may be 0.3, 0.2 i.e. 30% or 20% respectively for testing the model. The last parameter random_state sets the seed for a random generator so that the output is always the same.
The third line will print the following output

	Age	ExpectedSalary	India	Spain	U
4	38.000000	67000.0	0	1	0
9	44.000000	98000.0	0	0	0
1	28.000000	34000.0	0	0	0
6	39.777778	67000.0	1	0	0
7	53.000000	88000.0	0	1	0
3	45.000000	90000.0	0	0	0
0	28.000000	45000.0	1	0	0
5	54.000000	89000.0	0	0	1

7) Feature Scaling

This is the last step in the data pre-processing of machine learning. This method is used to standardize the independent variables of the dataset such that they are on the same scale and are comparable. Look in the dataset, Age and ExpectedSalary are independent variables and when they are compared, ExpectedSalary will dominate as it has much higher values than Age and will result in incorrect results. This issue will be reduced by performing feature scaling on the dataset. It is generally performed using two ways and these are:

a) New_value = (original_value – mean(original_value)) / standard deviation
b) New_value = (original_value – min(original_value)) / (max(original_value) – min (original_value))

For standardization, run the following code:

```
from sklearn.preprocessing import StandardScaler

st_x= StandardScaler()

x_train= st_x.fit_transform(x_train)

print(x_train)
```

Here, in the first line, StandardScaler class is imported for doing the standardization. The second line of the code indicates that an object of the StandardScaler class is created for independent variables. The third line shows that fir and transforms function is used for the training dataset but if the test dataset is there, you can directly apply the transform() function. There is no need to apply the fit_transform() function as it has been already performed in the training set. The fourth line prints the output for the x_train dataset and it is mentioned below and lies in the range of -1 and 1.

```
[[-0.34794225 -0.24192328 -0.57735027  1.73205081 -0.37796447]

 [ 0.29995021  1.1865761  -0.57735027 -0.57735027 -0.37796447]

 [-1.42776301 -1.76258391 -0.57735027 -0.57735027 -0.37796447]

 [-0.15597411 -0.24192328  1.73205081 -0.57735027 -0.37796447]

 [ 1.2717889   0.72576985 -0.57735027  1.73205081 -0.37796447]

 [ 0.40793229  0.8179311  -0.57735027 -0.57735027 -0.37796447]

 [-1.42776301 -1.25569704  1.73205081 -0.57735027 -0.37796447]

 [ 1.37977098  0.77185047 -0.57735027 -0.57735027  2.64575131]]
```

1.9 SELF-CHECK QUESTIONS

i. Which of the following is the best option for handling corrupted or missing data in the dataset?
   a) Use mean/mode/median values for filling missing values in the dataset
   b) In the case of categorical values, the unique category can be assigned to missing values.
   c) Missing columns or rows can be dropped.
   d) All of these

ii. List out the correct challenges that you may face while applying one-hot encoding on categorical variables.
   a) In the testing dataset, some categories of categorical variables are absent.
   b) Training and Testing datasets have an equal distribution of categories.
   c) Both options A and B
   d) Only A

iii. _____data is used for building models of machine learning.
   a) Training

b) Transfer
c) Missing
d) Corrupted

iv. The father of machine learning is _____.
a) Geoffrey Everest Hinton
b) Everest Gill
c) Hinton Geff
d) Javed Gill

v. Labelled data is used in what type of machine learning?
a) Supervised learning
b) Unsupervised learning
c) Reinforcement learning
d) None of these

vi. Choose the best correct option that best describes supervised learning.
a) The input data is labelled and machine learning algorithms learn to predict the output values from the given input data.
b) The input data is unlabeled and machine learning algorithms learn the inner structure from the given input data.
c) It can have both labelled as well as unlabeled data that suits output values.
d) All of these

vii. Choose the best correct option that best describes unsupervised learning.
a) The input data is labelled and machine learning algorithms learn to predict the output values from the given input data.
b) The input data is unlabeled and machine learning algorithms learn the inner structure from the given input data.
c) It can have both labelled as well as unlabeled data that suits output values.
d) All of these

viii. Give real-time examples of semi-supervised learning and reinforcement learning.

ix. What do you understand by Standardization methods?

x. Differentiate fit_transform() and transform() function.

xi. What is the impact when you drop rows and columns of missing data?

xii. Elaborate whether import the dataset and get the dataset are two different steps in data pre-processing?

1.10 SUMMARY

This module helps the students to understand that a particular problem lies in supervised learning, unsupervised, reinforcement, or semi-supervised learning area. A detailed discussion about the types of machine learning has been done. The students will be able to do data pre-processing on the datasets such that they are ready for performing machine learning algorithms. Deep discussions about how to import libraries and datasets, how to handle missing data, and how to split the data into training and testing data have been done. Different applications of machine learning have been discussed. This module helps the students in building up the basic blocks of machine learning that are required for doing hard problems or competitive problems at later stages.

1.11 UNIT END QUESTIONS

i. Select the type of supervised machine learning from the following options.
   a) Regression
   b) Classification
   c) Both a and b
   d) None of these

ii. Which of the following is the correct option while working on Unsupervised learning?
   a) Inputs are absent
   b) Output values are not present
   c) Both inputs and outputs are absent
   d) Output values are present

iii. At present which is the best language to work on machine learning programs?
   a) Python
   b) Java
   c) C
   d) HTML

iv. Is mathematics play an important role in advanced learning concepts of machine learning?
   a) Yes
   b) No

v. Explain in detail the use of train_test_split() function.

vi. Give insights about feature scaling with appropriate examples.

vii. List out the methods for encoding categorical variables along with suitable examples.

viii. Give details about the important libraries that are used for performing machine learning algorithms.

ix. Mention an example for each category of machine learning.

x. Discuss the future of machine learning in detail.

xi. Differentiate regression and classification in supervised learning.

REFERENCES

[1] https://www.geeksforgeeks.org/getting-started-machine-learning/

[2] https://searchenterpriseai.techtarget.com/definition/machine-learning-ML

[3] https://www.simplilearn.com/tutorials/machine-learning-tutorial/machine-learning-applications

[4] https://medium.com/app-affairs/9-applications-of-machine-learning-from-day-to-day-life-112a47a429d0

[5] https://www.analyticsvidhya.com/blog/2021/02/the-exciting-future-potential-of-machine-learning/

[6] https://www.upgrad.com/blog/data-preprocessing-in-machine-learning/

[7] https://analyticsindiamag.com/5-ways-handle-missing-values-machine-learning-datasets/

MACHINE LEARNING

UNIT II: REGRESSION

STRUCTURE

2.0 Introduction

2.1 Objectives

2.2 Linear regression

- **2.2.1 Linear Regression Code**
- **2.2.2 Regression Assumptions**
- **2.2.3 Linear Regression Code**

2.3 Multiple Linear Regression

2.4 Hypothesis Testing and p-value

- **2.4.1 Example p-value in Linear Regression**
- **2.4.2 Example p-value in Multiple Regression**
- **2.4.3 Key Terms Used in Regression**

2.5 Polynomial Regression

2.6 Comparison of Linear and Polynomial Regression

2.7 Self-check Questions

2.8 Summary

2.9 Unit End Questions

2.0 OBJECTIVES

- Knowledge about the Independent (explanatory) variables and a dependent (response) variable has been given along relation with each other has been figured out.
- How much independent variables will impact the dependent variables?
- Students will know about linear, multiple, and polynomial regression and even they can compare in which situation which regression technique will be used.
- Students will be able to do the python code for the regression themselves.
- Students can visualize the data by using python libraries.

2.1 INTRODUCTION

This module discusses regression in detail along with linear regression, multiple regression, and polynomial regression. Basic assumptions about the regressions have also been described. For every regression topic explanation, proper python code is given with output. To understand regression concepts, there is a need to understand p-value, hypothesis testing, and other model relevancy parameters. All these concepts have also been explained with suitable examples and python code. The focus of this module is on graduate students who want to learn how independent variables are related to dependent variables. The main target of this module is to inculcate the basics of different methods that are used for linear regression, multiple as well as polynomial regression. Students will have a clear understanding of dependent as well as independent variables along with regression equations. Moreover, for visualization of different models that have been developed by the students, the students have shown the use of python libraries that help the students to visualize the model results and improve the model performance after looking at the visualization results. The module is focused on regression problems and how python coding skills can be used to handle these problems. How the dataset is divided into training and testing set using python libraries has also been explored with many examples. At the section end, different types of regressions are compared visually with suitable python code so that students will have a clear understanding of the challenges and methods used in these regression techniques.

2.2 LINEAR REGRESSION INTRODUCTION

Linear regression is the key technique in supervised machine learning. The goal is to perform regression tasks. It predicts the value based on the independent variable. In other words, we can say, its task is to predict the response or dependent variable (Y) value when you are given one explanatory or independent variable (X). You are targeting to find the linear relationship among them [1]. One of the important things to remember is that Y should have continuous values.

Given below in equation 1 is the Hypothesis function:

$$Y = \$ + \$_1.X \quad (1)$$

where

X is an input variable

Y is output (continuous) variable

When the model is trained, the goal is to predict the value of the Y when you are given an X value. The model finds the value of $ and $_1$ for the best regression fit line. $ and $_1$ are intercept and coefficient of X respectively.

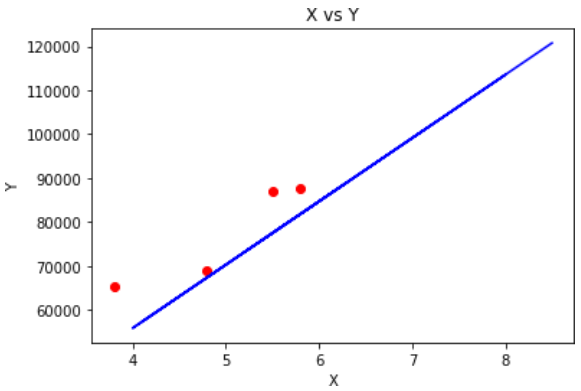


Figure 1: Linear Regression Example

Examples of linear regression:

a) Predict the market value of the shops.
b) Predict the stock price
c) Predicting the weight of a person
d) Person height prediction

2.2.1 Linear Regression Code Steps

It consists of seven steps and they are listed as:

a) Firstly, import the dataset
b) You can visualize the dataset using python commands (optional step)
c) After importing, split the dataset into training and testing sets. The training set should be between 70% to 95% of the dataset.
d) You can visualize training and testing set using python code (optional step)
e) Use the LinearRegression model and train the algorithm using the training set of the dependent variable and independent variable.
f) Now predict the output of the algorithm using the testing test of the independent variable of the training set.
g) Visualize the accuracy and different evaluation metrics using python code.

2.2.2 Regression Assumptions

It is a parametric approach. It means this approach is not suitable for prediction when its basic assumptions are not fulfilled. Any dataset must fulfill these assumptions before using regression. The assumptions of regression are [2]:

a) It assumes that a linear relationship should be present among the explanatory and response variable. It means if there is a unit change in the explanatory variable value then there must be only constant change in the response variable value. The effect of

the independent variable on the dependent variable is independent of other variables of the dataset.

If there is no linear relationship found then the model will result in doing wrong predictions. To check whether a linear relationship exists or not, use residual vs fitted plots.

b) No multicollinearity should exist between independent variables. In other words, it can be said that independent variables should not be correlated. If multicollinearity is present then model accuracy is drastically reduced and its presence is checked by using a scatter plot as well as you can check the VIF factor value and if it is greater than 4, it means multicollinearity is present.

c) Homoskedasticity: The constant variance must be there in error terms. If it is not there then it is called heteroskedasticity. The heteroskedasticity impacts the confidence interval as well as the model's performance. To check its presence you can use residual vs fitted values plot or Breusch-Pagan / Cook – Weisberg test or White general test

d) There should be a normal distribution of error terms. If it is not there, it also impacts confidence interval and it becomes difficult to determine coefficients correctly. You can check its presence using the QQ plot as well as the Kolmogorov-Smirnov test, Shapiro-Wilk test.

2.2.3 Linear Regression Code

```
# Firstly import libraries matplotlib for making plots and pandas for importing the dataset
import matplotlib.pyplot as mp
import pandas as pnds

# The below mentioned line is to import the dataset
dataset=pnds.read_csv(r'E:\Jagat Open State University\Machine learning\X_Y.csv')

print(dataset)  # The rows and columns of the dataset is printed
```

 # Output

```
   X    Y
0  3  3223
1  2  2123
2  4  4345
3  5  5498
4  6  6578
5  1  1234
6  7  7687
7  7  7569
8  4  4456
9  3  3354
10 8  8798
```

11 9 9897
12 8 8865
13 9 9900
14 2 2221
15 5 5489
16 9 9921
17 1 1090
18 4 4412
19 6 6599
20 6 6578
21 1 1234
22 7 7687
23 7 7569
24 4 4456
25 3 3354

dataset.shape # To know the number of rows as well as columns in the dataset

Output

(26,2)

dataset.describe()

Above line of code is used to get some statistical information

#Output

	X	Y
count	26.000000	26.000000
mean	5.038462	5543.730769
std	2.584272	2822.362805
min	1.000000	1090.000000
25%	3.000000	3354.000000
50%	5.000000	5493.500000
75%	7.000000	7657.500000
max	9.000000	9921.000000

Store the independent variable “x” values in “datax”

datax=dataset.iloc[:,:-1].values.reshape(-1,1)

```
# Store the dependent variable values “y” in “datay”

datay=dataset.iloc[:,1].values.reshape(-1,1)


# Below you find code that helps to divide the data, 75% of data for training and 25% data for test

from sklearn.model_selection import train_test_split

datax_tn, datax_tt, datay_tn,datay_tt = train_test_split(datax,datay,test_size=1/4, random_state=0)

# Now after splitting data, it’s time for training the algorithm using training data. Use LinearRegression class and fit() method for getting fit line

regr = LinearRegression()

reg.fit(datax_tn, datay_tn)

# Print intercept value

print(regr.intercept_)

#Print slope value

print(regr.coef_)
```

#Output

```
[40.14484536]
[[1092.29948454]]
```

```
datay_pred = regr.predict(datax_tt) # Predicting the data using testing data

dataf = pnds.DataFrame({'Actual': datay_test.flatten(), 'Predicted': datay_pred.flatten()})

print(dataf)
```

#Output

	Actual	Predicted
0	4345	4409.342784
1	6578	6593.941753
2	2221	2224.743814
3	1090	1132.444330
4	1234	1132.444330
5	9897	9870.840206
6	7687	7686.241237

```
regr.score(datax_tt,datay_tt)
```

#Output

0.999756848620213
Above output shows the accuracy of the model built

mp.scatter(datax_tt, datay_tt, color='gray')

mp.plot(datax_tt, datay_pred, color='red', linewidth=2)

mp.show()

#Output

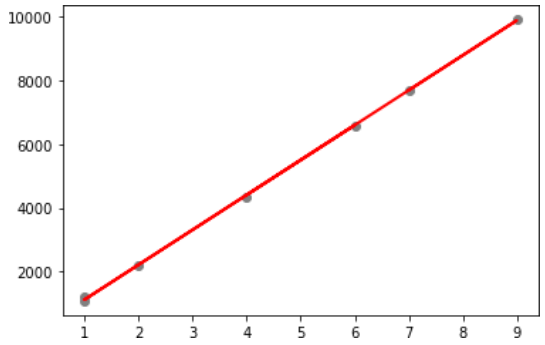


Figure 2: Linear Relationship Between X and Y

The above output in figure 2 shows the model is precise.

import sklearn.metrics as metrics

print(metrics.mean_absolute_error(datay_tt, datay_pred))

print(metrics.mean_squared_error(datay_tt, datay_pred))

import numpy as ny

print(ny.sqrt(metrics.mean_squared_error(datay_tt, datay_pred)))

#Output

36.420986745213085
2458.3050307077006
49.58129718661766
The algorithm performance can be known if you have the values for the above three metrics. [3].

2.3 MULTIPLE LINEAR REGRESSION

This is popularly also known as multivariate linear regression. It is having two or more two independent variables whereas, in linear regression, only one independent variable is there [4].

Hypothesis function for Linear Regression:

$Y = \$ + \$_1.X_1 + \$_2.X_2 + \ldots\ldots + \$_n.X_n + e$ (2)

where the dependent variable is Y, the intercept is $, regression line coeffcients are $\$_1$, $\$_2$, …. $\$_n$, and the error term is denoted as e in equation 2.

Suppose, there are two independent variables then the above equation 2 will become equation 3. The goal here is to estimate the values of $, $\$_1$, $\$_2$ such that predictions are close to actual values and it should be close to a perfect model.

$Y = \$ + \$_1.X_1 + \$_2.X_2 + e$ (3)

2.3.1 Multiple Regression Code

```
import matplotlib.pyplot as mp
import pandas as pnds
dataset=pnds.read_csv(r'E:\Jagat Open State University\Machine
learning\MultipleRegression.csv')
print(dataset)

#Output

    X1   X2  X3     Y
0    3  289  33  3223
1    2  212  22  2123
2    4  398  44  4345
3    5  489  55  5498
4    6  612  66  6578
5    1   98  11  1234
6    7  689  77  7687
7    7  713  72  7569
8    4  401  41  4456
9    3  298  31  3354
10   8  812  88  8798
11   9  899  99  9897
12   8  799  83  8865
13   9  923  92  9900
14   2  212  20  2221
15   5  515  51  5489
16   9  945  91  9921
17   1  101  10  1090
18   4  412  47  4412
19   6  654  69  6599
20   6  597  68  6578
21   1  102  16  1234
22   7  700  79  7687
23   7  745  74  7569
24   4  444  49  4456
```

25 3 299 39 3354

dataset.shape
#Output
(26, 4)

dataset.describe()
#Output

	X1	X2	X3	Y
count	26.000000	26.000000	26.000000	26.000000
mean	5.038462	513.769231	54.884615	5543.730769
std	2.584272	264.348907	26.992335	2822.362805
min	1.000000	98.000000	10.000000	1090.000000
25%	3.000000	298.250000	34.500000	3354.000000
50%	5.000000	502.000000	53.000000	5493.500000
75%	7.000000	709.750000	76.250000	7657.500000
max	9.000000	945.000000	99.000000	9921.000000

datax=dataset.iloc[:,0:3].values
datay=dataset.iloc[:,3].values
print(datax)

#Output
[[3 289 33]
[2 212 22]
[4 398 44]
[5 489 55]
[6 612 66]
[1 98 11]
[7 689 77]
[7 713 72]
[4 401 41]
[3 298 31]
[8 812 88]
[9 899 99]

```
 [  8 799  83]
 [  9 923  92]
 [  2 212  20]
 [  5 515  51]
 [  9 945  91]
 [  1 101  10]
 [  4 412  47]
 [  6 654  69]
 [  6 597  68]
 [  1 102  16]
 [  7 700  79]
 [  7 745  74]
 [  4 444  49]
 [  3 299  39]]
```

```
print(datay)
```

#Output

```
[3223 2123 4345 5498 6578 1234 7687 7569 4456 3354 8798 9897 8865 9900
 2221 5489 9921 1090 4412 6599 6578 1234 7687 7569 4456 3354]
```

```
from sklearn.model_selection import train_test_split
datax_tn, datax_tt, datay_tn,datay_tt = train_test_split(datax,datay,test_size=1/4,
random_state=0)

from sklearn.linear_model import LinearRegression
regr = LinearRegression()
regr.fit(datax_tn, datay_tn) # algorithm training

#Print intercept value
print(regr.intercept_)

#Print slope value
print("Coefficients:")
list(zip(datax, regr.coef_))
print(regr.coef_)
```

#Output

```
12.361448620703413
Coefficients:
[ 1.08705441e+03 -4.31213228e-01  5.03831029e+00]
```

```
datay_pred = regr.predict(datax_tt)
dataf = pnds.DataFrame({'Actual': datay_test.flatten(), 'Predicted': datay_pred.flatten()})
```

```
print(dataf)

#Output
   Actual    Predicted
0    4345  4410.641858
1    6578  6619.858684
2    2221  2195.819261
3    1090  1106.246421
4    1234  1112.578371
5    9897  9906.983124
6    7687  7717.919540

regr.score(datax_tt,datay_tt)*100
#Output
99.96784301426158

import sklearn.metrics as metrics
print(metrics.mean_absolute_error(datay_tt, datay_pred))
print(metrics.mean_squared_error(datay_tt, datay_pred))
import numpy as ny
print(ny.sqrt(metrics.mean_squared_error(datay_tt,  datay_pred)))

#Output
44.46457075920447
3251.1302170031745
57.018683052164356
```

2.4 HYPOTHESIS TESTING AND P-VALUE

This is one of the statistical procedures that is used to validate the results. It includes two statements, one is the null hypothesis and another one is the alternate hypothesis, they are notated as H_0 and H_a respectively.

For example:

H_0: There is no statistically significant relationship exists between variables X and Y.

H_a: There is a statistically significant relationship exists between variables X and Y.

The two things can happen now whether the null hypothesis is rejected or accepted. If we reject the null hypothesis, it means the alternate hypothesis is accepted and it means there exists a significant relationship between X and Y. This rejection and acceptance are done by looking at the p-value. The common threshold value for the p-value of 0.05. Here, 0.05 means that 5% of the time, the null hypothesis is falsely rejected.

Note: H_0 is rejected if the value of p is less than 0.05 and it means a significant relationship exists unless H_0 is accepted and it means a significant relationship does not exist between them. You can say that the p-value plays a very vital role in the acceptance and rejection of a hypothesis.

2.4.1 Example: p-value in Linear Regression

```
import pandas as pnds
dataset=pnds.read_csv(r'E:\Jagat Open State University\Machine learning\X_Y.csv')
print(dataset)

#Output
    X     Y
0   3  3223
1   2  2123
2   4  4345
3   5  5498
4   6  6578
5   1  1234
6   7  7687
7   7  7569
8   4  4456
9   3  3354
10  8  8798
11  9  9897
12  8  8865
13  9  9900
14  2  2221
15  5  5489
16  9  9921
17  1  1090
18  4  4412
19  6  6599
20  6  6578
21  1  1234
22  7  7687
23  7  7569
24  4  4456
25  3  3354

import statsmodels.api as smls
x_data = dataset['X']
y_data = dataset['Y']
z_data = smls.add_constant(x_data)
intermediate_model = smls.OLS(y_data, z_data)
```

```
final_model = intermediate_model.fit()
print(final_model.summary())
```

#Output

```
OLS Regression Results
==============================================================================
Dep. Variable:                      Y   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
Method:                 Least Squares   F-statistic:                 5.140e+04
Date:                Mon, 08 Nov 2021   Prob (F-statistic):           1.72e-41
Time:                        19:02:24   Log-Likelihood:                -143.25
No. Observations:                  26   AIC:                             290.5
Df Residuals:                      24   BIC:                             293.0
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         42.3564     27.162      1.559      0.132     -13.70      98.416
X           1091.8758      4.816    226.712      0.000    1081.936    1101.816
==============================================================================
Omnibus:                        1.045   Durbin-Watson:                   1.686
Prob(Omnibus):                  0.593   Jarque-Bera (JB):                0.942
Skew:                          -0.420   Prob(JB):                        0.624
Kurtosis:                       2.597   Cond. No.                         12.9
==============================================================================
```

Here, the p-value can be seen in P>|t| and it is 0.000 in front of the X variable which is less than 0.05. So, the null hypothesis is rejected or in other words, it can be said that the alternate hypothesis is accepted. It means there exists a significant relationship between the X and Y variable.

In the above program, the Ordinary least squares (OLS) method of linear regression is used and this method is in Python's Statsmodels module.

2.4.2 Example: p-value in Multiple Regression

```
import pandas as pnds
dataset=pnds.read_csv(r'E:\Jagat Open State University\Machine
learning\MultipleRegression.csv')
print(dataset)
```

#Output

```
   X1   X2  X3     Y
0   3  289  33  3223
1   2  212  22  2123
2   4  398  44  4345
3   5  489  55  5498
4   6  612  66  6578
5   1   98  11  1234
6   7  689  77  7687
7   7  713  72  7569
```

```
8   4  401  41  4456
9   3  298  31  3354
10  8  812  88  8798
11  9  899  99  9897
12  8  799  83  8865
13  9  923  92  9900
14  2  212  20  2221
15  5  515  51  5489
16  9  945  91  9921
17  1  101  10  1090
18  4  412  47  4412
19  6  654  69  6599
20  6  597  68  6578
21  1  102  16  1234
22  7  700  79  7687
23  7  745  74  7569
24  4  444  49  4456
25  3  299  39  3354
```

```
import statsmodels.api as smls
import numpy as np
x_data = np.column_stack((dataset['X1'], dataset['X2'], dataset['X3']))
y_data = dataset['Y']
z_data = smls.add_constant(x_data)
intermediate_model = smls.OLS(y_data, z_data)
final_model = intermediate_model.fit()
print(final_model.summary())
```

#Output

OLS Regression Results

Dep. Variable:	Y	R-squared:	1.000
Model:	OLS	Adj. R-squared:	0.999
Method:	Least Squares	F-statistic:	1.612e+04
Date:	Mon, 08 Nov 2021	Prob (F-statistic):	6.66e-37
Time:	19:24:05	Log-Likelihood:	-142.92
No. Observations:	26	AIC:	293.8
Df Residuals:	22	BIC:	298.9
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	.975]
const	34.8594	29.985	1.163	0.257	-27.326	97.045
x1	1085.1265	86.989	12.474	0.000	904.722	1265.531
x2	-0.2212	0.733	-0.302	0.766	-1.741	1.299
x3	2.8266	4.099	0.690	0.498	-5.675	11.328

Omnibus:	0.464	Durbin-Watson:	1.691
Prob(Omnibus):	0.793	Jarque-Bera (JB):	0.592
Skew:	-0.228	Prob(JB):	0.744
Kurtosis:	2.418	Cond. No.	4.01e+03

In the output, look at the values in P>|t| and the values are 0.000, 0.766, 0.498 for variables X1, X2, and X3 respectively. It shows that X1 has a significant relationship with Y whereas X2 and X3 do not have a significant relationship with Y.

2.4.3 Key Terms Used in Regression Outputs are

- **R-squared:** It is denoted as R^2 and popularly known as the coefficient of determination. It is a statistical measure that helps in knowing the variance proportion in the response variable that is explained by an explanatory variable or variables. For example, R^2 is 0.75 then it means model inputs explain third-fourth of the observed variation. The formula for the calculation of R^2 is mentioned in equation 1.

  $R^2 = 1- UV/TV$ (1)

  where UV is unexplained variation and TV is total variation

  R^2 also suffers from many limitations, it can not check whether there is biasness involved in predictions and coefficient estimates. It is also not able to check the adequacy of the model. A lower R2 value can come for a good model whereas a higher R2 value can come for an inadequate model.

- Adjusted R-squared: It is the revised form of R2 that is adjusted when more independent variables are included in the model. The value of adjusted R2 increases when added independent variables improve the model and it decreases when added extra independent variable decreases the model efficiency. But this is not the same with R2, it always increases when extra variables are added to the model.

- F-statistic: It helps in figuring out the overall significance of the model. It is calculated by finding out the ratio between model mean squared error and residuals mean squared error.

- Coef: In the output of the regression model, the coefficient is used for independent variables values and the constant value used in the regression equation.

- t: It denoted the t-statistic and it is calculated by doing a ratio of the difference between hypothesized and estimated parameters values to the standard error.

2.5 POLYNOMIAL REGRESSION

When nth degree polynomial relationship is present between explanatory and response variable, it is known as polynomial regression. To an extent, it is fairly like linear regression [6]. But here the non-linear relationship is there and the elements in this are the explanatory and conditional mean of the response variable.

2.5.1 Why there is a need for the polynomial regression

In polynomial regression, the assumption of all independent variables is independent is not satisfied.

When your regression line is curvilinear, then polynomial regression is used.

It provides a better approximation between the dependent variable and independent variable if the relationship is non-linear.

2.5.2 Example of Polynomial Regression with Code and Explanation

Suppose there are variables, one independent variable x_date tells the day number whereas y_corns tell the numbers of corns that are eaten on that particular day.

These two array values are mentioned as:

x_date = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,21,23]

y_corns = [102,92,81,79,62,61,59,55,61,63,68,69,74,75,78,79,88,93,97,102]

Firstly, going with model development, look at the scatter graph between x_date and y_corns using python code 1.

```
# CODE1

import matplotlib.pyplot as mp
x_date = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,21,23]
y_corns = [102,92,81,79,62,61,59,55,61,63,68,69,74,75,78,79,88,93,97,102]
mp.scatter(x_date, y_corns)
mp.show()

# Output (It is shown below in figure 3)
```

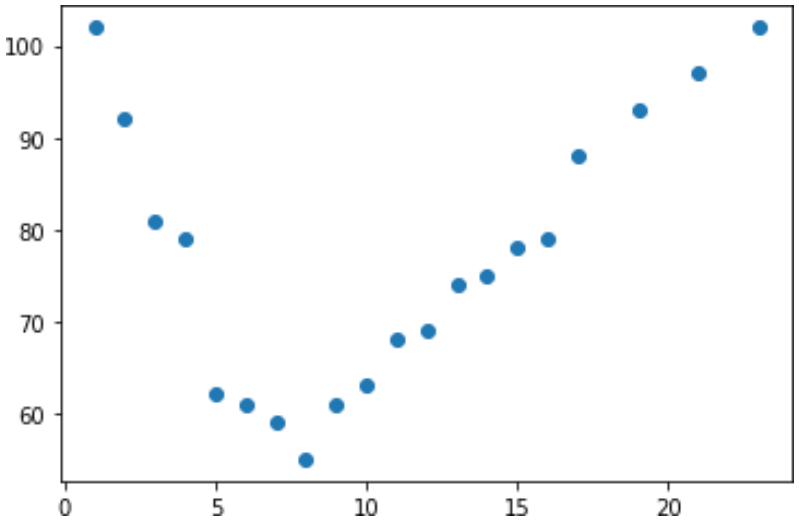


Figure 3: Scatter plot between x_date and y_corns

After looking at the scatter plot, draw the regression line between x_date and y_corns, and its code is shown in code 2 and its output in figure 4 [7].

```
# CODE 2
import numpy as ny
import matplotlib.pyplot as mp
from sklearn.metrics import r2_score

# Polynomial model is created with the help of numpy library
date_model_corns = numpy.poly1d(numpy.polyfit(x_date, y_corns, 3))

# Below line of code tells how regression line will be displayed i.e. it starts from 1 and ends
at 23.
reg_line = ny.linspace(1, 23, 100)

mp.scatter(x_date, y_corns) # This will help in drawing scatter plot

mp.plot(myline, mymodel(reg_line))
# This will help in drawing polynomial regression line

#Display the diagram:
mp.show()
#Output
```

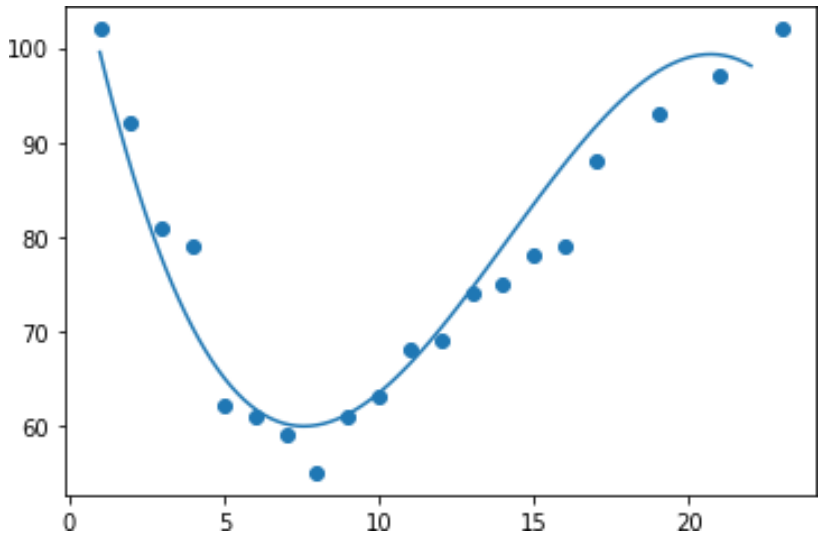


Figure 4: Polynomial Regression line between x_date and y_corns

After running code 2, it is important to know how well x_date and y_corns are related. In other words, it is important to know how well a relationship is associated with these variables. If no relationship exists between these two variables x_date and y_corns then this polynomial regression model will not be useful for predicting anything. So, R-Squared will be useful to know how well they are associated with each other. If r-squared values are nearer to 1, these variables are strongly related and if it is nearer to 0 then these variables are less or not related to each other. R-squared values lie between 0 to 1.

Use the following line at the end of CODE2 to find the r-squared value as sklearn helps in finding the r-squared value using the r2_score method.

```
print(r2_score(y_corns, date_model_corns(x_date)))
```

#Output

0.960913171163184

You can see in the above output, it lies nearer to 1 so a strong relationship exists between independent variable x_date and dependent variable y_corns.

The most important thing is now, to predict future values. It means how much corn will be eaten on day 13. Predict using the above polynomial regression model. To do this add the following lines at the end of CODE2.

```
eating_capacity = date_model_corns(12)

print(eating_capacity)
```

#Output

66.90762773392811

It can be seen prediction is nearer to the actual corns eaten on that day.

2.6 COMPARISON OF LINEAR AND POLYNOMIAL REGRESSION

```
import matplotlib.pyplot as mp
import pandas as pd
dataset=pd.read_csv(r'E:\Jagat Open State University\Machine
learning\polyregression.csv')
print(dataset)
```

#Output

	x_date	y_corns
0	1	102
1	2	92
2	3	81
3	4	79
4	5	62
5	6	61
6	7	59
7	8	55
8	9	61
9	10	63
10	11	68
11	12	69

```
12    13    74
13    14    75
14    15    78
15    16    79
16    17    88
17    19    93
18    21    97
19    23   102

x_date=dataset.iloc[:,:-1].values.reshape(-1,1)
y_corns=dataset.iloc[:,1].values.reshape(-1,1)
from sklearn.linear_model import LinearRegression
lr1 = LinearRegression()

lr1.fit(x_date,  y_corns)

from sklearn.preprocessing import PolynomialFeatures

py = PolynomialFeatures(degree = 4)
X_py = poly.fit_transform(x_date)

py.fit(X_py, y_corns)
lr2 = LinearRegression()
lr2.fit(X_py, y_corns)
#  For visualization of the results in a better way, the following lines of code is used
mp.scatter(x_date, y_corns, color = 'blue')

mp.plot(x_date, lr1.predict(x_date), color = 'red')
mp.title('Linear Regression')
mp.xlabel('x_date')
mp.ylabel('y_corns')

mp.show()

#Output (see linear regression model in figure 5)
```

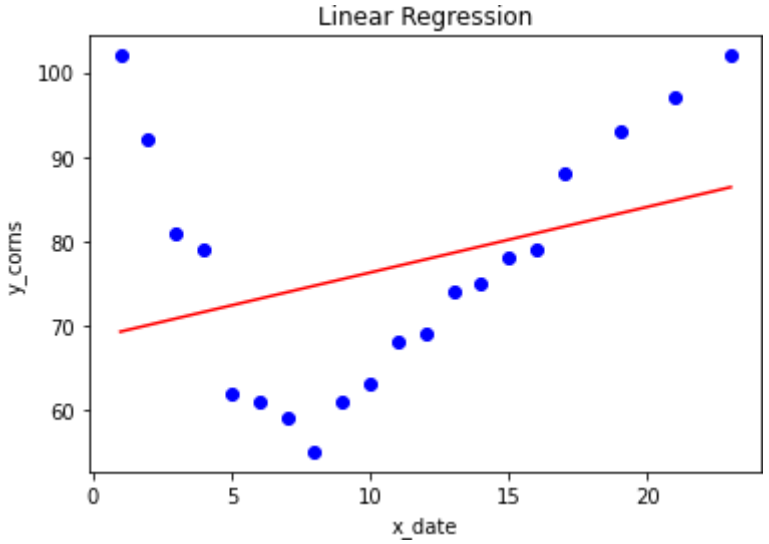


Figure 5: Linear Regression Model

For visualization of the results in a better way, the following lines of code is used

```
mp.scatter(x_date, y_corns, color = 'blue')

mp.plot(x_date, lr2.predict(py.fit_transform(x_date)), color = 'red')
mp.title('Polynomial Regression')
mp.xlabel('x_date')
mp.ylabel('y_corns')

mp.show()
```

#Output (See figure 6 for polynomial regression model)

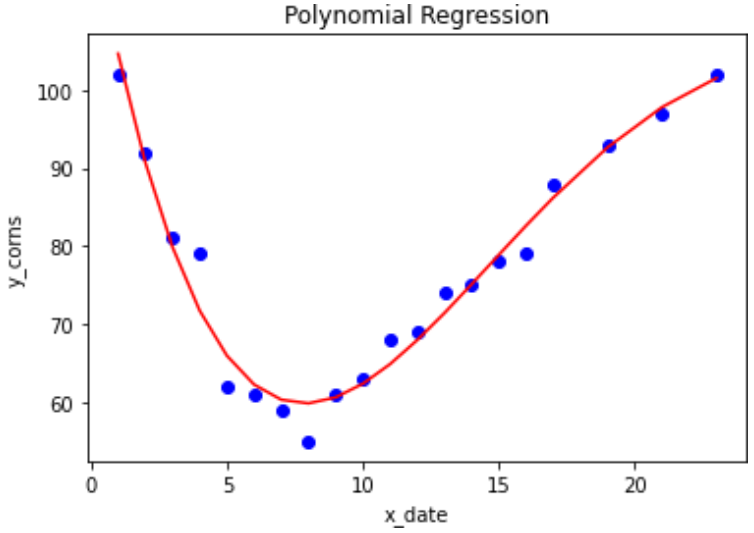


Figure 6: Polynomial Regression Model

From figure 5 and figure 6, it is clear that the polynomial regression model is best suited for an approximation when there is a non-linear relationship among variables.

2.7 SELF-CHECK QUESTIONS

i. Graphically draw the linear regression. Find out the R^2 value and p-value for the following data.

Xinput_data	Yinput_data
2	18
4	16
6	20
8	24
10	22
12	25
14	27

ii. Find out the regression equation of Yinput_data on Xinput_data and also estimate Yinput_data when Xinput_data=14 for the following data:

Xinput_data	Yinput_data
4	16
8	31
12	46
16	63
20	78
24	90
28	100

iii. Differentiate null hypothesis and alternate hypothesis with an example.

iv. What is the use of independent variables while calculating R^2 value and adjusted R^2 value?

v. Choose the correct option from the following choices that is not an assumption of linear regression
   a) Linear relationship
   b) Variance should be constant
   c) Multicollinearity
   d) Residuals should be normally distributed

vi. Categorical predictors impact _____and continuous predictors impact _____of the regression line.
   a) Intercept, slope
   b) Adjusted R^2, intercept
   c) p-value, intercept
   d) None of these

vii. Is to predict the winner of a cricket game, simple linear regression is sufficient? (True or False)

viii. Choose the correct option that does not represent the simple linear regression formula:
   a) Profit_output = a_input * Sales
   b) Profit = a_input * Sales + b_constant
   c) Profit = a_input* Sales + b_input * Cost
   d) None of these

ix. A simple linear regressor is created by the class _____for coding in python.

x. Choose the correct option from the following that does not represent multiple regression formula:
   a) Profit = a_input* Sales + b_input * Cost + c_constant
   b) Profit = a_input* Sales + b_input * Cost + c_input * RawMaterial
   c) Profit = a_input* Sales + b_input * Cost + c_input * RawMaterial + d_constant
   d) Profit = a_input* Sales + b_input * Cost ^ 2

xi. Is the class that is used to create the linear regressor is same for multiple linear regressors in Python coding? Tell the answer is True or False and also mention the class name.

xii. Linear regression belongs to supervised machine learning or not? Also, mention the linear regression equation with a detailed explanation of each parameter used in the regression equation.

xiii. Select the best option from the following that shows the impact of outliers on the linear regression.
   a) It is sensitive towards outliers
   b) It is insensitive towards outliers
   c) Difficult to say
   d) None of these

xiv. Look at the scenario, a linear regression model is underfitting the data and choose the best option that fits into that scenario
   a) Removal of some independent variables
   b) Addition of some independent variables
   c) Introduce polynomial degree variables in the regression model
   d) It can be any one of the above cases

xv. Create a linear regression model using python code for the below-mentioned data

Xinput_data	**YOutput_data**	**Xinput_data**	**YOutput_data**	**Xinput_data**	**YOutput_data**
23	90	40	167	24	90
30	135	33	140	34	120
6	26	9	38	5	30
18	70	14	67	8	29
15	62	17	70	35	145
12	50	10	44	37	139
21	84	20	80	31	156
32	126	29	120	39	118
11	44	28	118	40	160

16	61	19	70	42	168

2.8 SUMMARY

After completing this module thoroughly, students will be able to do python code for different regression techniques like linear, multiple, or polynomial regression. Students are aware of the assumptions that are used for model development using regression. Students will know which different classes will be used for different types of regressors. They can do a comparison between different types of regression and they can do hypothesis testing. Students are inculcated with visualization python code such that they can visualize the data to improve the model relevancy. Students will know key parameters that are used to know for the model relevancy like R-squared, Adjusted R-squared, p-value, etc. They can do hypothesis testing by themselves and can decide after evaluating the parameters of regression models. They can understand whether the explanatory variable should be added to the model or it should be dropped from the model. How many explanatory variables can be added that explains the response variable sufficiently? Students can also check the overall significance of the model by looking at the F-statistic too. This module has inculcated the python coding skills that are required for working on regression problems and regression applications. They can also handle real-time problems.

2.9 UNIT END QUESTIONS

i. Is F-statistic is used to know the overall significance of the model. If yes, explain its suitability in knowing the overall model significance.

ii. Write the number of steps using statsmodels.api, when you want regression straight line fit through data points.

iii. What do you mean by heteroscedastic in linear regression? When this condition is available can you run your model using simple linear regression?

iv. Compare heteroscedastic with homoscedastic in linear regression with suitable examples.

v. Mention in detail when you want to reject the Null hypothesis and accept the Null hypothesis, the role of the p-value.

vi. As linear regression has an assumption that a linear relationship exists between explanatory variables and a single response variable. What is the meaning of this assumption? Choose the correct and best option from the following given choices

   a) All the explanatory variables when combined can't calculate the response variable.
   b) All the explanatory variables when combined can calculate the response variable.
   c) All response variables when combined linearly then they can predict explanatory variables.
   d) All explanatory variables when summed together then they can calculate the explanatory variable.

vii. The two models have been mentioned below with their R-squared value on the training set as well as testing set. Both models have worked on the same data set and the second model has more root mean squared error than the first model.

R-squared values for the first model on the training set and testing set are 0.90 and 0.58 respectively.

R-squared values for the second model on the training set and testing set are 0.76 and 0.73 respectively.

Which of the above-mentioned models is the better one?

viii. Let suppose if one of the featured independent variables (Ind_Var) is well explained by the other independent variables, this means Ind_Var matches has

a) High VIF value
b) High p-value
c) Low p-value
d) Low VIF value

ix. Write the applications of linear regression with suitable examples.

x. How you can handle the problem of overfitting and underfitting the data in the linear regression model?

xi. What do you understand by VIF and explain in detail how it is a useful parameter to get the information for variables.

xii. Create a linear regression model using python code for the below-mentioned data

Xinput_data	YOutput_data
2	10
4	18
6	24
8	32
10	35
12	42
14	42
16	48
18	45
20	50
22	44
24	48
26	39
28	42

xiii. Create a multiple regression model using python code for the below-mentioned data. Also, tell the model is fit or not.

X1	X2	X3	Y
2	217	35	5600
2	212	22	6588
4	456	45	4567
5	544	57	5690
3	344	35	3543

1	89	13	1876
6	689	66	6687
6	645	62	6569
4	465	49	4876
4	468	41	4359
8	856	82	8198
8	790	84	8235
8	787	86	8165
9	999	94	9999
9	912	92	9991
9	915	94	9989
9	949	97	9888
4	498	52	4900
5	512	55	5000
6	600	67	6543
6	639	69	6545

xiv. Let's suppose for the goodness of fit in the linear regression problem, you are considering the R-squared measure. Then, suddenly you add a new feature in the linear regression model and the whole model is trained again. Then, choose the best option from the following given choices.

a) If by adding this independent variable in the model, the R-squared value increases then this added variable is significant.
b) If by adding this independent variable in the model, the R-squared value decreases then this added variable is not significant.
c) It can not be said by just considering the R-squared value whether the independent variable that you are adding in the linear regression model will have importance or not.
d) None of the above-mentioned choices

xv. List down the application of regression in detail. Also, mention the parameters that impact regression model relevancy. Every parameter must be elaborated.

xvi. Which library of python is used for dividing the datasets into training part and testing part. Make a python code for this concept.

REFERENCES

[1] https://www.geeksforgeeks.org/ml-linear-regression/

[2] https://www.analyticsvidhya.com/blog/2016/07/deeper-regression-analysis-assumptions-plots-solutions/

[3] https://www.kdnuggets.com/2019/03/beginners-guide-linear-regression-python-scikit-learn.html

[4] https://medium.com/machine-learning-with-python/multiple-linear-regression-implementation-in-python-2de9b303fc0c

[5] https://www.geeksforgeeks.org/ordinary-least-squares-ols-using-statsmodels/

[6] https://www.geeksforgeeks.org/python-implementation-of-polynomial-regression/

[7] https://www.w3schools.com/python/python_ml_polynomial_regression.asp

UNIT III: CLASSIFICATION

STRUCTURE

3.0 Objectives

3.1 Introduction

3.2 Logistic regression

3.3 K-Nearest neighbours

3.4 Support vector machine

3.5 Naïve bayes

3.6 Decision tree classification

3.7 Random forest classification

3.8 Self-check questions

3.9 Summary

3.10 Unit-end questions

3.0 OBJECTIVES

- Understand various classification and regression algorithms.
- Understand the working and implementation of classification algorithms.
- Learn the pros and cons of machine learning classification algorithms.

3.1 UNIT INTRODUCTION

This module targets various classification algorithms along with their working, applications, advantages, disadvantages and implementation in python. This module also covers the real-time examples related to these classification algorithms. The module targets graduate students who are eager to learn and implement various machine learning classification models. The classification algorithms focus to classify the dataset based on many different parameters like its class, group, features, attributes, etc. The main aim of the module is to give a basic understanding of these classification algorithms along with their implementation in a python programming language.

3.2 LOGISTIC REGRESSION

This statistical analysis technique is commonly utilised in prognostic analytics and development, as well as machine learning applications. Using a logistic regression equation to estimate probabilities, is employed in analytical software to comprehend the link between one or more independent factors and the dependent variable. This form of analysis can assist you in predicting the chances of an occurrence or a decision occurring. For example, a person wants to know the possibility of a user visiting the company website. So, with the help of Logistic regression, a person can able to recognize the behaviour of a user like the number of times the user visits the website, which product a user look into, from which site a user is coming to the company's website and many more. The logistic model assists the person to decide that what type of users are accessing their website and as a result better can be taken for promotion of company and website [1].

The logistic regression model is used to forecast the probability of a specific event, such as pass/fail, up/down, dead/alive, ill/healthy, fresh/stale. A logistic function is used by logistic regression to present a binary dependent variable in its most primary form, while there are many more complicated extensions available. A binary logistic model mathematically contains two dependent variables with two possible values like win/loss, these dependent variables represent by two different indicators which are known as labelling like "1" and "0", where "1" stands for the win and "0" stands for loss. By computing the likelihood of each element of the set, logistic regression is used to classify elements of a set into two categories (binary classification) [1].

a. Importance of Logistic regression:

Predictive models created using the logistic regression method can have a favourable impact on any company or organisation. Because these models aid in the understanding of linkages and the prediction of consequences, one can use them to make better decisions. For example, a restaurant owner can able to decide the probability of customers visiting on a particular day depending on the previous pattern. With this, the restaurant owner can analyse and decide the

number of dishes they can prepare accordingly without any failure. Along with this, the logistic regression analysis method can be used in medicine to estimate the likelihood of sickness or illness in a given population, allowing for the implementation of preventative care. By monitoring buyer behaviour, businesses can identify trends that lead to improved employee retention or produce more profitable products. This form of analysis is used in the corporate world by data scientists, whose purpose is to evaluate and comprehend complicated digital data [2].

b. Types of Logistic regression:

The three types of logistic regression are:

a. **Binomial logistic regression:** In this type of logistic regression the variable target can have two possible values of indicators like “healthy” or “diseased”, “1” or “0”, “Victory” or “Loss”.

b. **Multinomial logistic regression:** This type of logistic regression deals with three or more unordered target variables like “Disease X” vs “Disease Y” vs “Disease Z”.

c. **Ordinal logistic regression:** It works with ordered categories of target variables. For example, a class grade can be categorized as “low grade”, “medium grade” and “high grade”.

c. Prediction using logistic regression:

To make predictions with a logistic regression model, it's as simple as inserting numbers into the equation and computing the outcome. For example, let’s assume that a model has been developed that will predict whether a person is a boy or girl based on their height (imaginary).

Assuming the height of a person as 160cm.

The learnable coefficients are c0=-100 and c1=0.6. Using the following equation we can able to get the resultant value which will predict that whether the person is a boy or a girl.

S(x) = e ^ (c0+c1*x) / (1 + e ^ (c0+c1*x)), where x is the given height of a person and S(x) is the resultant predicted value.

So, by applying the above formula using the given values we will get,

S(x) = e ^ (-100+0.6*160) / (1 + e ^ (-100 + 0.6*160))

S(x) = 0.01831 / 1.01831

S(x) = 0.01798

We can apply the probabilities directly in practice. To have a clear answer, the resultant values have been reduced to binary values.

0 if S (male) < 0.5

1 if S (female) > 0.5

So, in this case, as the resultant value is less than 0.5 it gets compressed to 0 which predicts it as a “male”.

d. Applications of Logistic regression:

In a variety of domains, including machine learning, medical domains, and the social science field, logistic regression is applied. Along with this many medical measures that are used to determine a patient's severity were developed using logistic regression. Based on the patient's observed features, logistic regression can be used to forecast the likelihood of developing a disease. Logistic regression is also applied in engineering for the prediction of a particular system to be a success or failure, in marketing it is used as a customer behaviour prediction and in the discipline of natural language processing this approach is used for sequential data regression. Some of the real-time applications of logistic regression have been shown below [3].

a. Scoring of credit in finance
b. Text editing
c. Medical database
d. Hotel booking system
e. Gaming

e. Implementation of Logistic regression with python:

To implement Logistic regression with python, there are four basic steps to follow which are stated below.

3.2.5.1 Import Packages, functions and Classes

To use some of the pre-defined functions, we need to import some of the packages and classes.

Code:

```
import matplotlib.pyplot as pl_t

import numpy as np_o

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import classification_report, confusion_matrix
```

3.2.5.2 Fetching Data

For the implementation of the approach, one needs to have some data on which the code will process, so in this case, we have created two arrays for input and output.

Code:

```
a = np_o.arange(9).reshape(-1, 1)
```

b = np_o.array([0, 0, 0, 1, 1, 1, 1, 1, 1])

3.2.5.3 Development and Training of the Model

You can design and define your categorization model once you have the input and output ready. You'll use an instance of the class LogisticRegression to represent it.

Code:

mode = LogisticRegression(solver='liblinear', random_state=0)

After development of model, the next part is to train the model, which is done with the help of .fit() function.

Code:

mode.fit(a, b)

3.2.5.4 Model Evaluation

After development and successful training of the defined model, the next and last step is to evaluate and test the model based on resultant predictions with the help of the .predict_probe() and .predict() function.

Code:

mode.predict_proba(a)

mode.predict(a)

3.3 K-NEAREST NEIGHBOURS

K-Nearest neighbours (K-NN) is a classification technique based on the non-parametric method developed by Joseph Hodges and Evelyn fix. This technique is also used regression in which the k closest training examples in a data collection are used as input and the output is dependent on whether the K-NN technique is applied for regression or classification. Class membership is the result of K-NN classification. An object is assorted by the utmost vote of its neighbours, with the object allocated to the most familiar class with its k closest neighbours. While in K-NN regression, the output is the average of the values of k nearest neighbours. In K-NN classification, the function is only estimated sectionally, and all computation is postponed until the function is assessed. As this process depends on classification distance, normalising the training data can greatly enhance the system's accuracy if the features represent various physical units. A productive approach for both classification and regression is to designate weights to the support of the neighbours so that the nearer neighbours give more to the mean than the remoter neighbours. Along with this, the neighbours are selected from a group of objects from which the class is known [4].

Some of the properties of K-NN are discussed below.

a. The K-NN method is one of the most fundamental machine learning algorithms. It is based on the supervised learning technique.
b. The K-NN technique presumes that fresh data and current cases are exact and assigns the new case to the category that is closest to the existing categories.
c. K-NN stocks all the accessible data and performs classification based on likeness.
d. K-NN is designed for both regression and classification, but in most cases, K-NN is applied for the classification of the dataset.
e. K-NN never takes any pre-assumed data.
f. The KNN method merely saves the information during the training phase, and when new data is received, it is placed in a category that is alike to the newly developed data.

Example of K-NN:
Suppose we have an image of a person that seems similar to a male or a female, but we don't know whether it's a male or female. We can utilise the KNN method for this recognition because it is based on a measure of similarity. Our KNN model will compare the data set to the male and female pictures and categorise it as a male or a female based on the most similar attributes.

3.3.1 Need of K-NN:
One of the most extensively used learning algorithms and one of the fundamental classification methods is the KNN approach. Its purpose and necessity is to use a database of data points separated into several groups to predict the categorization of a new sample data point [4].

3.3.2 Working of K-NN:
The working of K-NN can be easily understood with the help of a simple example. Figure 1 shows the distribution of yellow circles (YC), green triangles (GT) and a pink diamond (PD).

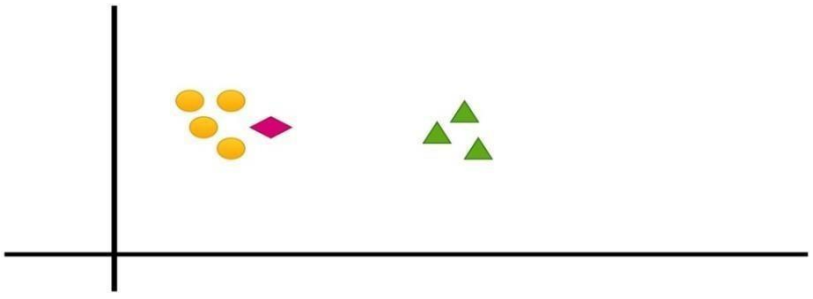


Fig 1. Distribution of YC, PD and GT

Using a K-NN approach we need to find the class of PD which can belong to either YC or GT. The "K" in the KNN algorithm stands for the nearest neighbour from whom we want to take a vote. So, in this case, we take the value of K as 4, hence a circle with PD will be made covering the nearest four data points as shown in figure 2.

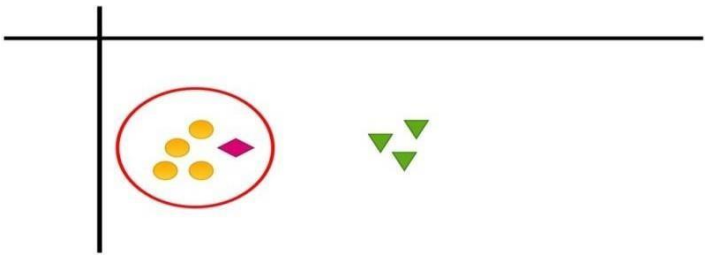


Fig 2. Class of PD using K-NN

The four locations closest to PD are all YC. As a result, we can confidently assert that the PD goes to the YC classification. The winner was obvious in this case, as YC won all four votes from the nearby neighbourhoods. The parameter K must be carefully chosen in this technique [5].

3.3.3 Selecting value of K in K-NN

The following are some of the points which one should keep in mind while selecting the value of K.

a. There is no one-size-fits-all method for determining the ideal value for "K," so we'll have to experiment with a variety of options to find the best one.
b. K=1 or K=2 is an extremely low value for K that might be noisy and cause outlier effects in the model.
c. Large values for K are desirable, but they may cause problems.

3.3.4 Advantages

The following are some of the advantages of K-NN.

a. In the case of a large amount of training data, K-NN proved to be more effective and efficient.
b. It is more reliable in the case of the noisy dataset.
c. The implementation of K-NN is very simple.

3.3.5 Disadvantages

Some of the disadvantages of K-NN has been mentioned below.

a. The value of K must constantly be determined, which might be challenging at times.
b. The calculation cost is high since the distance between the data points for all of the training samples must be calculated.

3.3.6 Applications of K-NN

K-NN has been applied in a variety of fields like,

a. Finance
b. Face recognition
c. Recommendation system for websites
d. Text analytics
e. Security systems.

3.3.7 Implementation of K-NN:

In the implementation of k-NN, the following steps have been executed.

a. Using scikit-learn package classifier and dataset has been imported.
b. Development of feature variable.
c. Dividing of data into training and testing.
d. Creation of K-NN model with the help of neighbour values.
e. Training of the model using dataset.
f. Prediction of the values

Code:

```
# Import classifier and dataset
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

# Dataset
i_data = load_iris()

# Developing feature variables
a = i_data.data
b = i_data.target

# dividing data into training and testing
a_tr, a_te, b_tr, b_te = train_test_split(a, b, test_size = 0.3 random_state=50)

k = KNeighborsClassifier(n_neighbors=6)

knn.fit(a_tr, b_tr)

# Predict on dataset which model has not seen before
print(k.predict(a_te))
```

3.4 SUPPORT VECTOR MACHINE (SVM)

Among various supervised learning methods, SVM is one of the well-known techniques that helps for both classification and regression problems. The SVM algorithm's purpose is to find the optimum line for categorising n-dimensional space into groups so that in the future, further data points can be easily placed in the correct group The ideal choice boundary is known as a hyperplane. The hyperplane has been created by selecting the extreme points which are known support vectors and the technique is known as a support vector machine [6].

3.4.1 Working of SVM

An SVM model is a representation of separate classes in a hyperplane in multi-dimensional space. To minimise the inaccuracy, SVM will iteratively create the hyperplane. The goal of

SVM is to divide datasets into classes so that a maximum marginal hyperplane can be discovered [6].

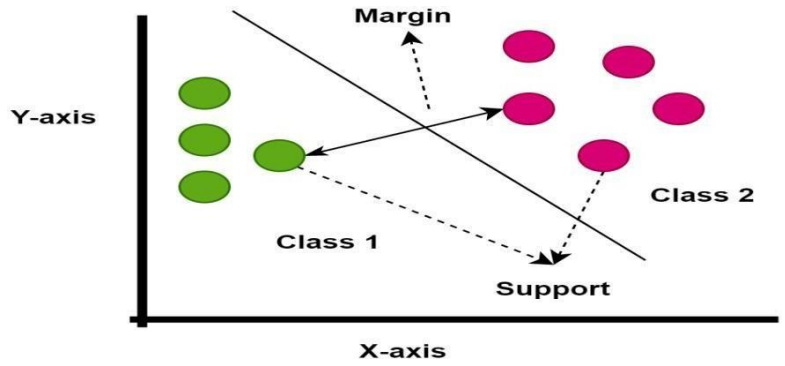


Fig 3. SVM concepts

Some of the important concepts of SVM are as follows:

a. Support vectors: The data points closest to the hyperplane are called support vectors. A dividing line will be drawn between these data points.
b. Figure 3 shows a hyperplane. It is a decision plane or space in which several objects of distinct classes are separated.
c. Margin: It's the distance between two lines on a data point with distinct classifications on a closet. It is possible to calculate the perpendicular distance between the line and the support vectors. A large margin is considered a good margin, whereas a tiny margin is considered a bad margin.

3.4.2 Implementation of SVM in Python

The SVM algorithm utilizes kernels to transform a low-dimensional input space into a higher-dimensional data space. This is known as the kernel trick. The kernel makes SVM more powerful, flexible, and accurate by adding more dimensions to non-separable problems.

Code:

```
import numpy as n_pl
import matplotlib.pyplot as p_lt
from sklearn import svm, datasets
# import data
ir = datasets.load_iris()
X_data = ir.data[:, :2] # Taking only two features.
Y_data = ir.target

# Developing an instance of SVM for fitting out the data.
C_param = 1.0
S_vc = svm.SVC(kernel='linear', C=1,gamma=0).fit(X_data, Y_data)

# Developing mesh for plotting
```

```
X_mi, X_ma = X_data[:, 0].min() - 1, X_data[:, 0].max() + 1
Y_mi, Y_ma = X_data[:, 1].min() - 1, X_data[:, 1].max() + 1
h = (X_ma / X_mi)/100
xg,yg  = n_pl.meshgrid(n_pl.arange(X_mi, X_ma, h),
 n_pl.arange(Y_mi, Y_ma, h))
p_lt.subplot(1, 1, 1)
Zp = S_vc.predict(n_pl.c_[xg.ravel(), yg.ravel()])
Zp = Zp.reshape(xg.shape)
p_lt.contourf(xg, yg, Z, cmap=plt.cm.Paired, alpha=0.8)
p_lt.scatter(X_data[:, 0], X_data[:, 1], c=y, cmap=p_lt.cm.Paired)
p_lt.xlabel('Sepal length')
p_lt.ylabel('Sepal width')
p_lt.xlim(xg.min(), xg.max())
p_lt.title('SVC with linear kernel')
p_lt.show()
```

3.4.3 Advantages

The following are the advantages of SVM.

a. When there is a clear separating line, it works well.
b. It is very efficient in the case of high dimensional spaces.
c. This approach works effectively when the number of dimensions is higher than the sample number.
d. Because the decision function only employs a subset of training data, it is memory efficient.

3.4.4 Disadvantages

Some of the disadvantages of SVM are mentioned as follows.

a. It does not work well when we have a huge data collection because the required training lapse is longer.
b. In the case of overlapping of the target class and additional noise in a dataset, it performs poorly.
c. Probability estimates are generated via a time-consuming five-fold cross-validation method that SVM does not give directly.

3.5 NAÏVE BAYES

It is a classification technique that assumes that a particular feature in a class does not correlate with the presence of other features in that class. As a result, they are based on Bayes' Theorem. For example, if a vegetable is red and round, with a diameter of 2 inches it is known as a tomato. Even though these features are dependent on one another or the presence of other characteristics, they are still important, each of them contributes to the possibility that this vegetable is a tomato, and this is a reason we named it 'naïve' [7].

The Naive Bayes model is easy to build and works well with large data sets. Because of its simplicity, Naive Bayes is known to outperform even the most powerful classification algorithms.

3.5.1 Working of Naïve Bayes Algorithm

The working of the Naïve Bayes algorithm can be understood with the help of an example. We need to classify that whether an athlete is qualified for the game or not based on he is healthy or not, so for that, we need to follow some steps [8].

Step 1: Develop a frequency table based on the dataset.

Step 2: A likelihood table will be created by finding the unhealthy probability as 0.30 and the probability of qualified players as o.60.

Table 1. Dataset table

Player status	**Qualified**
Healthy	Yes
Unhealthy	No
Healthy	Yes
Healthy	Yes
Unhealthy	No
Healthy	Yes
Healthy	Yes
Unhealthy	No
Unhealthy	No
Healthy	Yes

Table 2. Frequency Table

Frequency Table		
Player status	**No**	**Yes**
Healthy	0	6
Unhealthy	4	0
Grand total	4	6

Table 3. Likelihood table

Likelihood table				
Player status	**No**	**Yes**		
Healthy	0	6	=6/10	0.60
Unhealthy	4	0	=4/10	0.40
All	4	6		
Probability	4/10=0.40	6/10=0.60		

So, by referring to Table 3., we can conclude that player with healthy status has a higher probability to get qualified for the game. Based on attributes that use the same approach naïve Bayes predict the likelihood of various classes. This method is commonly used for the classification of text and challenges involving several classes.

3.5.2 Advantages of Naïve Bayes

a. Predicting the class of test data sets is fast and easy. It also does well in multiclass prediction

b. When the assumption of independence is met, the Naive Bayes classifier outperforms other models such as logistic regression. This also cuts down on the amount of data needed for training.

c. Comparatively to numerical variables, it performs well when the input variables are categorical.

3.5.3 Disadvantages of Naïve Bayes

a. If the test data set contains the categorical variable whose category was not included in the training data set, a probability of 0 (zero) will be assigned to the model and prediction will not be generated. This is known as "Zero Frequency".

b. Naive Bayes is a lousy estimator, as a result, the probability output should be handled with caution.

3.5.4 Applications of Naïve Bayes algorithm

a. Naive Bayes is a classifier that learns quickly and eagerly. As a result, real-time forecasting could be possible.

b. This method is known for its ability to predict a wide range of classes. This predictor may predict the chance of many target variable classes, making it a multi-class predictor.

c. In collaboration with Naive Bayes Classifiers, Collaborative Filtering creates Recommendation Systems that helps in predicting whether a user will be interested in a given resource or not.

3.6 DECISION TREE CLASSIFICATION

The decision tree technique is widely used for the classification of classes and making predictions. It is a flowchart like structure with every internal node present with an attribute test, the result of the test represents by branch and the class label represents every leaf node [9].

There are three parts of the decision tree which are mentioned as follows and also shown in figure 4.:

a. Node: Node consists of a vale of a particular attribute.
b. Edge: It is a connecting link that connects the two nodes.
c. Leaf node: It is the terminal or end node which predicts the outcome.

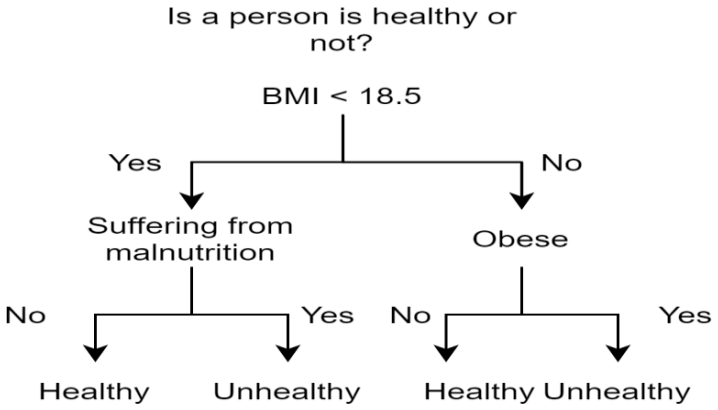


Fig 4. Decision tree

Classification trees and regression trees are two types of decision trees which are discussed as follows [10].

3.6.1 Classification Trees

It is a logical representation of a decision tree that gives the class of an object. This is an iterative method that involves separating the data into partitions and then further splitting it up on each branch.

3.6.2 Regression Trees

Regression trees are the type of decision trees with target variables taking continuous values. For example price of a share of a company or the length of a customer stay in a hotel.

3.6.3 Divide and Conquer Algorithm

Recursive partitioning is used to build decision trees due to which is also known as the divide and conquer algorithm. Divide and conquer is an approach to data analysis where data is divided into subgroups, which are repeatedly subdivided into smaller subgroups, and so on until the algorithm determines that the data within the subgroups is homogeneous or another

stopping criterion is satisfied. There are four basic steps of the divide and conquer algorithm which is discussed as follows [10].

a. Select a root node and create a branch of all the possible outcomes.
b. Subdivide instances into groups. Each branch extending from the node has its own.
c. For each branch, repeat the process recursively, utilising just the instances that make it to the branch.
d. If all of a branch's instances share the same class, the recursion should be stopped.

3.6.4 Advantages of Decision Trees

a. Economical to implement
b. Unknown records are classified extremely quickly.
c. For many simple data sets, accuracy is equivalent to that of other classification algorithms.

3.6.5 Disadvantages of Decision Trees

a. Very easy to overfit.
b. Splits on features with a large number of levels are frequently favoured by decision tree models.
c. Large trees can be hard to understand, and their decisions might be surprising.

3.6.6 Applications of Decision Trees

a. Finance analysis
b. Medical diagnosis and analysis
c. Feature detection in biomedical engineering
d. particle recognition in physics

3.6.7 Python Code for Decision Tree

The below is the step by step python implementation of the decision tree:
Step 1: Importing all the important libraries.

```
import pandas as p_lt
import numpy as n_plt
import matplotlib.pyplot as p_lt
import seaborn as n_s
%matplotlib inline#for encoding
from sklearn.preprocessing import LabelEncoder#for train test splitting
from sklearn.model_selection import train_test_split#for decision tree object
from sklearn.tree import DecisionTreeClassifier#for checking testing results
from sklearn.metrics import classification_report, confusion_matrix#for visualizing tree
from sklearn.tree import plot_tree
```

Step 2: Loading of the dataset.

```
ds = n_s.load_dataset('iris')
ds.head()
```

Step 3: Pre-processing of the dataset.

```
t = ds['species']
ds1 = ds.copy()
ds1 = ds1.drop('species', axis =1)
F=ds1
#label encoding
l = LabelEncoder()
t = l.fit_transform(t)
t
g=t
```

Step 4: Dividing the data into training and testing

```
f_tr, f_te, g_tr, g_te = train_test_split(f , g, test_size = 0.3, random_state = 42)print("Training split input- ", X_train.shape)
print("Testing divide input- ", f_te.shape)
```

Step 5: Testing the model:

```
D_tree.fit(f_train,g_train)print('Decision Tree Classifier developed')
g_pr = D_tree.predict(f_test)
print("Classification- ", classification_report(g_test,g_pr))
```

3.7 Random Forest Classification

The idea behind random forests is to build a multitude of decision trees during training time to perform classification and regression. It Is majorly used for classification. As we all know, a forest is made up of trees, and more trees equal a healthier forest. The random forest technique, on the other hand, creates decision trees from data samples, extracts predictions from each, and then votes on the best alternative [11].

3.7.1 Working of Random Forest Classification

The initial stage in Random Forest is to join N decision trees, followed by predicting the outcomes of each tree formed in the first phase [12]. The working flow of random forest is explained using the following steps and figure 5.

Step 1: N data points selected from the training set.

Step 2: Creating decision trees that are linked to a set of N data points.

Step 3: Choosing the K decision trees which should be developed.

Step 4: Perform steps 1 and 2 again.

Step 5: Find each decision tree's forecasts for new data points, assign the new data points to the category with the most votes, and forecast the outcome.

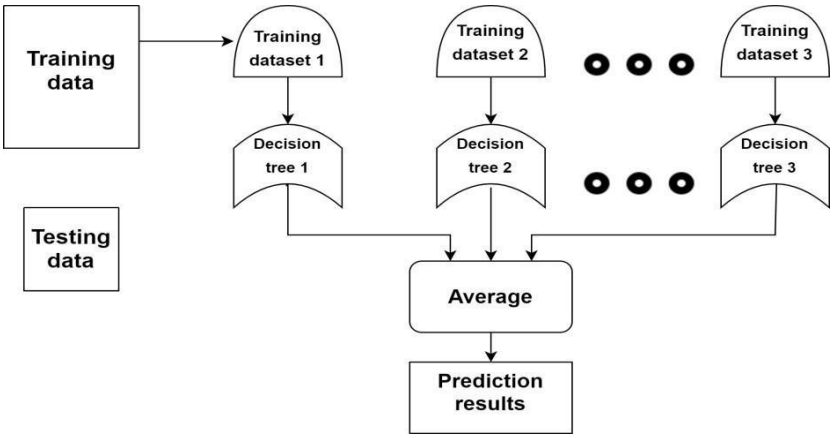


Fig 5. Random forest classification

3.7.2 Presumptions before Working With Random Forest

As numerous trees combine for forecasting the class of dataset in a random forest, some outcomes anticipate with the decision trees correctly while others may not. But if all the trees collectively get combined will be resulted in an accurate prediction [11]. Therefore two assumptions for correct random forest classifier are shown below:

a. To get the correct and accurate outcome feature variable of the dataset mast must have actual values.
b. Low correlation must be there for each tree's forecast.

3.7.3 Advantages of the Random Forest

a. It can perform both regression and classification tasks.
b. Capable of controlling high dimensional large datasets
c. Stop overfitting problems and increase the accuracy of the model.

3.7.4 Disadvantages of the Random Forest

a. Contains complex calculations
b. In some cases, it results in sub-optimal decision trees.

3.7.5 Applications of the Random Forest

Some of the applications of random forest are mentioned below:

a. Fraud detection in banking
b. Disease prediction in medical fields.
c. Share market prediction
d. Sentiment analysis
e. Product suggestion on an e-commerce site.

3.7.6 Implementation of Random Forest in Python

Implementation of random forest contains basic steps which are discussed below with python code.

Step 1: Pre-processing of data:

```
# import libraries
```

```
import numpy as n
import matplotlib.pyplot as m
import pandas as p
#import datasets
d= p.read_csv('user_data.csv')
#Extraction of variables
xr= d.iloc[:, [2,3]].values
yr= d.iloc[:, 4].values
# dividing the dataset into training and testing.
from sklearn.model_selection import train_test_split
xr_tr, xr_te, yr_tr, yr_te= train_test_split(xr, yr, test_size= 0.25, random_state=0)
#feature extractio
from sklearn.preprocessing import StandardScaler
s_xr= StandardScaler()
xr_tr= s_xr.fit_transform(xr_tr)
xr_te= s_xr.transform(xr_te)
```

Step 2: Fitting the algorithm to training dataset:

```
from sklearn.ensemble import RandomForestClassifier
classifier= RandomForestClassifier(esti= 10, crit="entropy")
classifier.fit(xr_tr, yr_tr)
```

Step 3: Predicting the result using testing dataset:

```
yr_pred= classifier.predict(xr_te)
```

Step 4: Creation of Confusion matrix:

```
from sklearn.metrics import confusion_matrix
c= confusion_matrix(yr_te, yr_pred)
```

3.8 SELF-CHECK QUESTIONS

Que 1: Can we say that Logistic regression is a type of supervised machine learning algorithm?

A. True

B. False

Que 2: The main and only purpose of logistic regression is regression?
A. True
B. False

Que 3: To best fit the data, which approach should be applied in logistic regression?

A. Maximum likelihood
B. Jaccard distance

C. Least Square error
D. None of the above

Que 4: Which of the following assessment criteria cannot be used to compare the output of logistic regression with the target?

A. Accuracy
B. AUR-ROC
C. Mean square error
D. None of these

Que 5: The assumption for logistic regression is?

A. The logit of the result variable has a linear connection with the continuous predictor factors.
B. Continuous predictor variables and the outcome variable have a linear connection.

C. A linear relationship between observations.
D. None of the above

Que 6: The k-NN technique does more computing during the test phase than during the training phase.

A. True
B. False

Que 7: In k-NN, which of the following distance metrics can't be used?

A. Jaccard
B. Tanimoto
C. Manhattan
D. All of the above

Que 8: Which of the following statements concerning the k-NN algorithm is correct?

A. It's a classification tool.
B. It’s a regression tool
C. It can be useful for both classification and regression

Que 9: Which of the following statements concerning the k-NN algorithm is correct?

A. If all of the data has the same scale, k-NN performs substantially better.
B. k-NN works effectively when the number of input variables (p) is small, but it struggles when the number of inputs is large.
C. k-NN does not presume the functional form of the problem to be solved.
D. All of the above

Que 10: What should be the value of k that will reduce the cross-validation accuracy while leaving out one?

A. 5
B. 3
C. Both have equal accuracy
D. None of these

Que 11: In terms of the SVM, what do you mean by generalisation error?

A. The distance between the support vectors and the hyperplane
B. The SVM's ability to predict outcomes for data that hasn't been observed.
C. The maximum amount of error that an SVM can tolerate.
D. None of these

Que 12: Which of the following is true when the C parameter is set to infinite?

A. If one exists, the best hyperplane is the one that totally isolates the data.
B. The data will be separated by the soft-margin classifier.
C. Both A and B
D. None of these

Que 13. O(n2) is the simplest temporal complexity for training an SVM. What dataset sizes aren't best suited for SVMs, based on this fact?

A. Small dataset
B. Large dataset
C. Medium dataset
D. None of these

Que 14: When some features are absent, how can we conduct Bayesian classification?

A. The missing values are assumed to be the mean of all values.
B. We overlook the features that aren't present.
C. Over the missing features, we combine the posterior probability.
D. None of the above

Que 15: Which of the following statements about a Decision Tree is TRUE?

A. The classification issue statement is the only one for which a decision tree is appropriate.
B. As we progress along a decision tree, the entropy of each node reduces.
C. Only numeric and continuous attributes can be used in a decision tree.
D. Entropy influences purity in a decision tree.

Que 16: When building a decision tree, how do you pick the proper node?

A. An entropy-rich attribute
B. A property with high entropy and information gain.
C. An attribute with the least amount of information gained.
D. An attribute with the greatest amount of knowledge gain.

Que 17: When it comes to correlation and covariance, which of the following statements is FALSE?

A. A zero correlation does not always imply that the variables are independent.
B. The values of correlation and covariance are the same.
C. The sign of covariance and correlation is always the same.
D. The standardised version of Covariance is Correlation.

Que 18: When it comes to Deep Learning and Machine Learning algorithms, which of the following statements is FALSE?

A. Deep Learning algorithms can handle a large amount of data with ease.
B. Unstructured data is best suitable for Deep Learning algorithms.
C. Deep Learning algorithms need a lot of computing power.
D. In both ML and DL systems, feature extraction must be done manually.

Que 19: Which of the following statements concerning the ensemble algorithms Random Forest and Gradient Boosting are true?

A. For a classification problem, both methods can be employed.
B. For regression tasks, both techniques can be used.
C. For regression, Random Forest is used, whereas, for classification, Gradient Boosting is used.
D. Both A and B

Que 20: In a random forest, how many decision trees are there?

A. several single trees, each based on a random subset of the training data
B. A random forest is a set of decision trees that aren't connected in any way.
C. Both A and B
D. None of the above

3.9 SUMMARY

This module will help students to understand various machine learning classification algorithms along with their real-time applications, pros and cons. Along with this working and implementation of these classification algorithms has been discussed in detail. In every section of the module, a basic introduction of the module followed by its working, assumptions taken for the algorithm to work, its pros and cons, a real-life application which covers the areas or fields where that particular algorithm has been used, and at last step by step implementation of the algorithms in a python programming language has been discussed in detail. This module will assist the student to understand the basic concept of classification in machine learning with the help of various machine learning classification algorithms.

3.10 UNIT END QUESTIONS

Que 1: What is logistic regression?
Que 2: Explain different types of Logistic regression?
Que 3: Explain Properties of K-Nearest Neighbour?
Que 4: Explain the assumption before working with random forest classification?
Que 5: Describe the advantages, disadvantages and applications of decision trees?
Que 6: Explain the working of the Support vector machine?
Que 7: Explain different parts of the decision tree with the help of an example?
Que 8: Explain different types of decision trees?
Que 9: Implement Decision tree classification with the help of python programming language?
Que 10:Describe the working of Random forest classification and implement it in a python programming language?

References

[1] https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc

[2] https://en.wikipedia.org/wiki/Logistic_regression

[3] https://machinelearningmastery.com/logistic-regression-for-machine-learning

[4] https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning

[5] https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering

[6] https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code

[7] https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/

[8] https://www.geeksforgeeks.org/naive-bayes-classifiers

[9] https://www.geeksforgeeks.org/decision-tree/

[10] https://www.geeksforgeeks.org/decision-tree/

[11] https://www.javatpoint.com/machine-learning-random-forest-algorithm

[12] https://medium.com/swlh/random-forest-classification-and-its-implementation-d5d840dbead0

MACHINE LEARNING
UNIT IV: CLUSTERING

STRUCTURE

4.0 Objectives

4.1 Introduction to Classification

4.2 K-means clustering

4.3 K-means random initialization trap

4.4 Selecting the number of clusters

4.5 Hierarchical clustering

4.6 Self-check questions

4.7 Summary

4.8 Practice questions

4.0 OBJECTIVES

- Understand various clustering algorithms.
- Understand the working and implementation of clustering algorithms.
- Learn the pros and cons of machine learning clustering algorithms.
- Learn and understand the pre-requisite for the implementation of clustering algorithms.
- Students can able to code and run these algorithms on their respective datasets.

4.1 UNIT INTRODUCTION

This module covers a variety of clustering algorithms, including how they operate, what they're used for, what they're good for, what they're bad for, and how to implement them in the python programming language. This section also includes real-world examples of clustering algorithms in action. Along with this it also covers the detailed explanation of each algorithm with stepwise diagrammatic representation which will help the reader to understand the concept deeply and properly. A deep explanation has been given in the case of the k-means cluster and hierarchical clustering concept with step-wise python code and their respective clustering output. Dataset for implementation has been self-developed and uploaded to understand the algorithm more clearly. Graduate students who want to understand and use various machine learning classification models can benefit from this programme. Clustering algorithms data sets or data points depend on a variety of factors such as closeness or distance between data points. The module's main goal is to provide a fundamental overview of various clustering techniques, as well as their implementation in Python.

4.2 K-MEANS CLUSTERING

K-Means Clustering is an unsupervised learning method used in machine learning (ML) and data science to handle clustering challenges. Its goal is to make k number of clusters using t observations, and every observation belongs to the cluster with the nearest average which acts as the cluster's prototype. In the k-means clustering algorithm, the variance within a cluster is minimized, only the geometric median results in the smallest squared errors. It splits the unlabelled dataset into various clusters. Already defined clusters are specified by k, that must be produced during the process; if K=5, five clusters will be created, and if K=4, four clusters will be created. It's an iterative method for dividing an unlabelled dataset into k clusters, with each dataset belonging to a single group with identical characteristics or traits. In this way, we can cluster the data into different types of bundles and find on our own without training the type of groups in unlabelled datasets. This approach is based on centroid that is every cluster has its centroid. This technique's major purpose is to reduce the total distance between data points and the clusters to which they belong [1].

The technique takes unlabelled data as input, separates it into a k clusters, and the procedure will be repeated until no better clusters are found. In this algorithm, the value of k should be already fixed. The following are the tasks that are performed by the k-means clustering algorithm.

a. Determines the optimal value for K centre points.

b. The k-centre that is closest to each data point is assigned to it. Data points that are close to a given k-centre create a cluster.

The unsupervised k-means algorithm is connected to the k-nearest neighbour classifier, a well-known supervised ML classification technique that is frequently confused with k-means due to its name. Using the cluster centres generated by k-means, the 1-nearest neighbour classifier is used to categorise incoming data into existing clusters. This is known as the Rocchio algorithm or nearest centroid classifier [2].

4.2.1 History of k-means Clustering:

Following the idea of Steinhaus which came in 1956, James MacQueen used the name "k-mean" in the year 1967. Stuart Lloyd from bell labs proposed a modulation technique of pulse-code in 1957. Along with this in 1965 W. Forgy proposed the same method which was later published due to that reason the algorithm is known as the Lloyd-Forgy algorithm [3].

4.2.2 Working of k-means Clustering Algorithm

This technique's primary purpose is to lower the sum of distances between data points and the clusters to which they belong; the working of k-means clustering involves the set of steps which are explained as follows [1].

Step 1: Select the number of clusters by selecting the specific value of k.

Step 2: Select any point randomly which will act as a centroid or centre.

Step 3: Assign or select the data point nearest to the centroid which will result in the pre-defined k-clusters.

Step 4: Placing the new centroids by calculating the variance.

Step 5: Perform Step 3 again to get the new k-clusters which will result by reassigning the nearest data points to the centroids.

Step 6: Perform Step 4 again in case of any reassignment or finish the process.

Step 7: The model is developed and ready.

The demonstration of k-means clustering has been shown in figure 1.

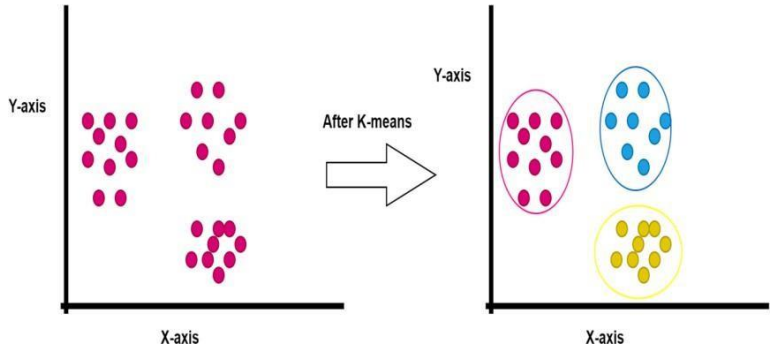


Fig 1. k-means clustering

Now let's understand the above-mentioned step by step working with the help of an example and graphical plots.

Considering that we have two different variables with the name d1 and d2. The x-y scatter or graphical plot of these two variables has been shown in figure 2.

1. To recognise the dataset and place it into various clusters, we'll use the number k of clusters, i.e., K=2. That is, we will attempt to divide these datasets into two distinct clusters.

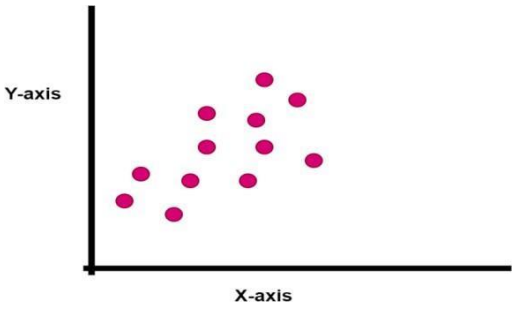


Fig 2. Graphical plot of two variables

2. To form the cluster, we'll need to pick a random k points or centroid. These points could be from the dataset or from anywhere else. As a result, we've chosen the two points below as k points, even though they're not in our dataset as shown in figure 3.

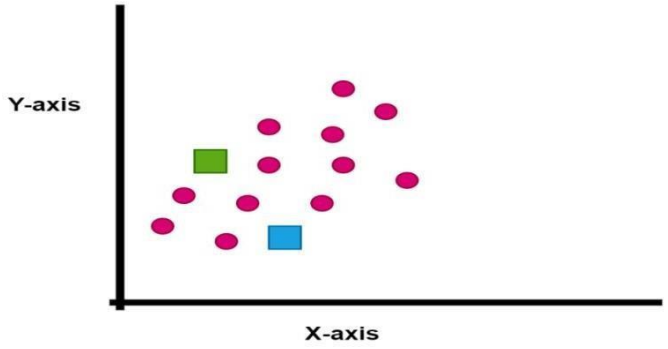


Fig 3. Graphical plots with k points

3. Now we'll assign each scatters plot data point to the nearest K-point or centroid. We'll figure it out using the arithmetic we've learned about calculating distances between two spots. As a result, we'll draw a line connecting both centroids, as shown in figure 4.

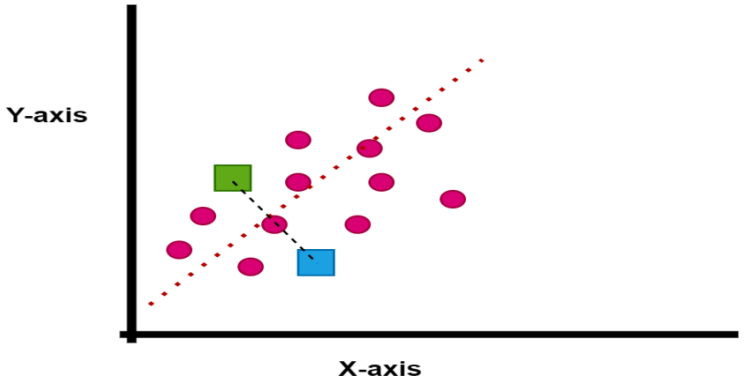


Fig 4. The connecting line between centroids

4. Figure 4 shows that points on the line's left side are near to the f1 or green centroid, Points on the right side of the line are close to the blue centroid. To make them easier to see, we'll paint them green and blue, as seen in Figure 5.

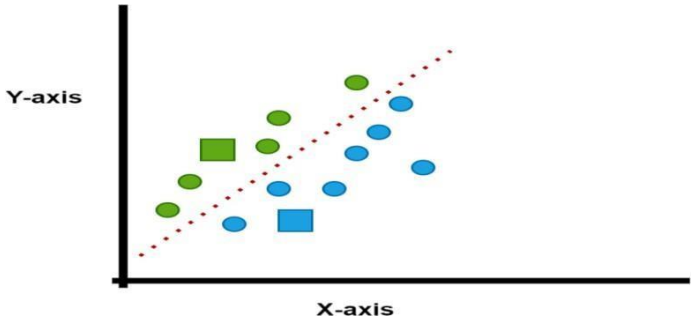


Fig 5. Reassigning colour to the data points

5. We'll repeat the process by selecting a new centroid because we need to discover the closest cluster. We'll compute the centre of gravity of these centroids to determine new centroids, as shown in figure 6.

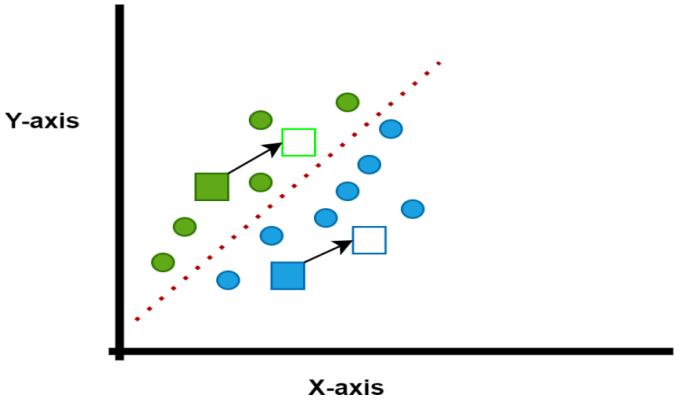


Fig 6. Selection of new centroid

6. Then, for each data point, we'll reassign it to the new centroid. We'll go through the same steps as before to identify a median line. Figure 7 depicts the median.

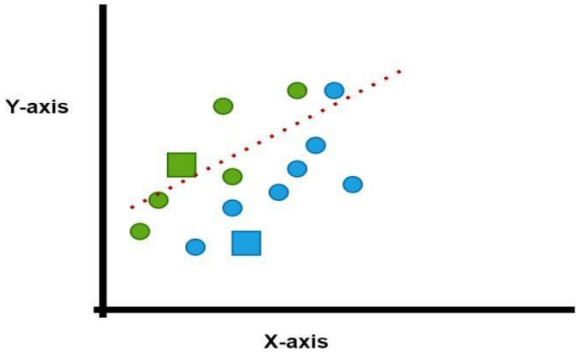


Fig 7. Reassignment of new centroid

7. From figure 7 it has been clear that on the left side of the line one blue point is there and three green points are on the right which will result in the assignment of new centroids. We'll return to step-4, which is determining new centroids or K-points, after reassignment is complete, which will lead us to again follow all the steps again and result in the result of the fully developed model with two final clusters as shown in figure 8.

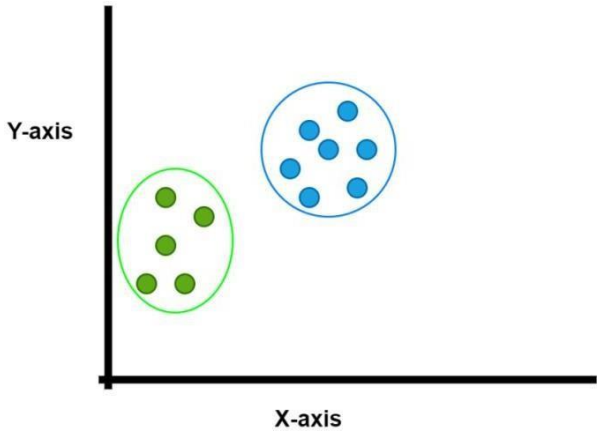


Fig 8. Fully developed model

4.2.3 Implementation of k-means clustering algorithm in python:

To implement the algorithm in the python programming language the following set of steps has been followed.

Step 1: Creation of dataset

For a simple implementation, a demo dataset has been developed using a .csv file which was later imported using python. For demonstration, the developed dataset has been shown in Table 1.

Table 1. Employee dataset

Index	EmpID	Gender	Age	Salary (In k rupees)	Work score
1	1	Male	20	25	10
2	2	Female	33	40	5
3	3	Male	21	25	6
4	4	Male	40	40	3
5	5	Female	28	25	7
6	6	Female	19	15	6
7	7	Male	26	25	9
8	8	Male	22	25	4
9	9	Male	35	40	9
10	10	Male	41	40	8

Step 2: Uploading and pre-processing of the data

Code:

```
import numpy as n_p
import matplotlib.pyplot as m_atlip
import pandas as p_do
from google.colab import files
up = files.upload()
import io
f_r = p_do.read_csv(io.BytesIO(up['exc.csv']))
```

Extraction of independent variable:

```
x_r = f_r.iloc[:, [3, 4]].values
```

Step 2: By using elbow method we will find the optimum cluster numbers

Code:

```
from sklearn.cluster import KMeans
wcss= []  #Initializing the list for the values of WCSS
#Using for loop for iterations from 1 to 10.
for i in range(1, 11):
    k_m = KMeans(n_clusters=i, init='k-means++', random_state= 42)
    k_m.fit(x_r)
    wcss.append(k_m.inertia_)
m_atlip.plot(range(1, 11), wcss)
m_atlip.title('Graph')
m_atlip.xlabel('No. of clusters(k)')
m_atlip.ylabel('wcss')
m_atlip.show()
```

Output:

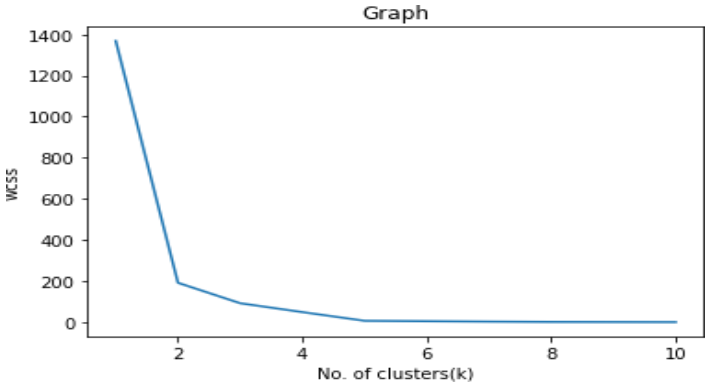


Fig 9. The output of Step 2

Step 3: Training the model

In step 3 the developed model will be trained using the training data.

Code:

```
k_m = KMeans(n_clusters=5, init='k-means++', random_state= 42)
y_p= k_m.fit_predict(x_r)
```

Step 4: Cluster visualization

The clusters must then be seen as the final stage. Because our model has five clusters, we'll look at each one separately using the below-mentioned code and figure 10.

Code:

```
m_atlip.scatter(x_r[y_p == 0, 0], x_r[y_p == 0, 1], s = 100, c = 'blue', label = 'Cluster 1') #for first cluster
m_atlip.scatter(x_r[y_p == 1, 0], x_r[y_p == 1, 1], s = 100, c = 'green', label = 'Cluster 2') #for second cluster
m_atlip.scatter(x_r[y_p== 2, 0], x_r[y_p == 2, 1], s = 100, c = 'red', label = 'Cluster 3') #for third cluster
m_atlip.scatter(x_r[y_p == 3, 0], x_r[y_p == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4') #for fourth cluster
```

```
m_atlip.scatter(x_r[y_p == 4, 0], x_r[y_p == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')
#for fifth cluster
m_atlip.scatter(k_m.cluster_centers_[:, 0], k_m.cluster_centers_[:, 1], s = 300, c = 'yellow', la
bel = 'Centroid')
m_atlip.title('Clusters of Employee')
m_atlip.xlabel('salary (k rupees)')
m_atlip.ylabel('Work Score (1-10)')
m_atlip.legend()
m_atlip.show()
```

Output:

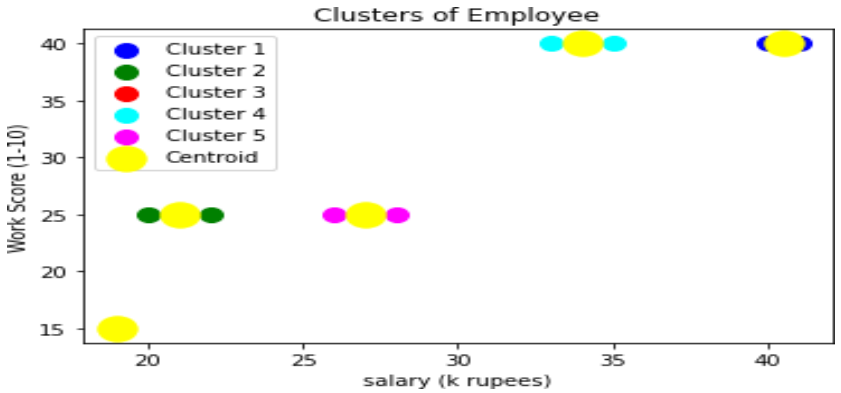


Fig 10. Cluster visualization

4.2.4 Advantages of k-means Clustering

1. Simple Implementation
2. Handles big datasets
3. Surety of Convergence.
4. Warm up the positions of centroids.
5. Works with same efficiency in new situations.
6. It can be applied to elliptical clusters of various forms and sizes.

4.2.5 Disadvantages of k-means Clustering

1. Selection of vale of k manually
2. Initial values dependencies
3. When data clusters are of different sizes and densities, k-means has problems clustering them.
4. Distance-based similarity measures converge to a fixed value as dimensions increases between any two cases.

4.2.6 Applications of k-means Clustering

1. Academic scoring system
2. In medicine, K-means are utilised to develop smarter medical decision support systems, especially in the treatment of liver illnesses.
3. Search engines helps to give efficient results using clustering.
4. Wireless sensor networks can able locate cluster heads using clustering.

4.3 K-MEANS RANDOM INITIALIZATION TRAP

The K-means algorithm has a problem called the random initialization trap. When the cluster centroids formation are defined by the user in the random initialization trap, inconsistency can be created, which can lead to incorrect clusters being generated in the dataset. As a result, the random initialization trap may occasionally prevent us from forming the correct clusters [4].

Each run of k means produces a different WCSS when the centroids are randomly initialised. Clustering is suboptimal when the centroids are chosen incorrectly. We use K-means++ to tackle the problem of inaccurate centroids by selecting the centroids as far as possible at initialization. The concept is to use centroids to construct different cluster centres to achieve optimal clustering and converge quickly [5].

Now, let's understand the concept of the random initialization trap with the help of an example. Consider figure 11 with data points on the graph, now we want to make clusters of this dataset using k-means clustering approach based on the attributes or features. Now, according to the concept of k-means clustering, the clustering will be performed based on centroids and according to them, the algorithm will generate different clusters.

Trial 1:

Based on figure 11 we choose three different centroids in the dataset as shown in figure 12, which will generate the final model with three different clusters, shown in three different colours in figure 13.

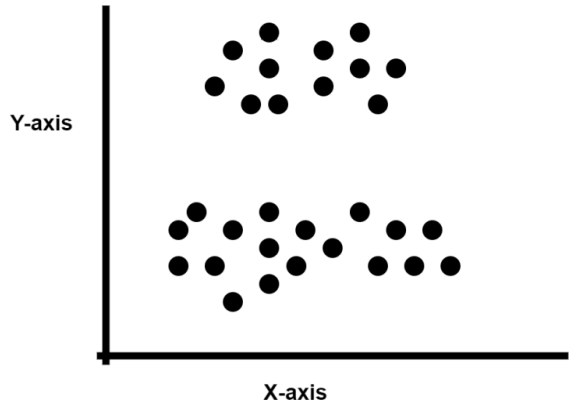


Fig 11. Data points shown graphically

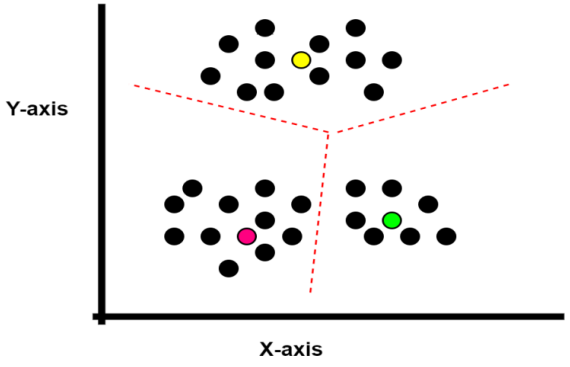


Fig 12. Data points with centroids in trial 1

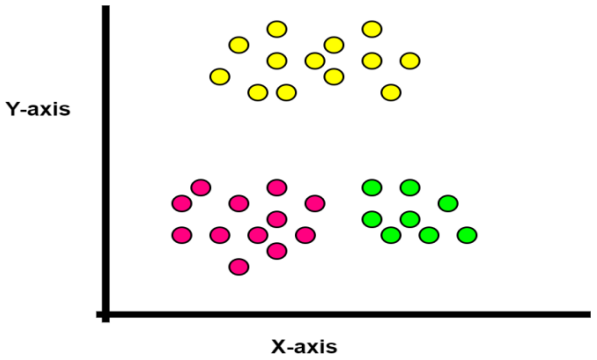


Fig 13. Formation of three different clusters in trial 1

Trial 2:

Now considering another situation in which we take another set of centroids different from Trail 1 as shown in figure 14, now that will generate the final model different from the earlier but logically correct as shown in figure 15.

Now from the above example, it has been observed that on the same dataset, we could receive multiple model outputs. It is a circumstance in which a different set of clusters is created When the K-means algorithm is given a different set of centroids, it becomes inconsistent and reliable.

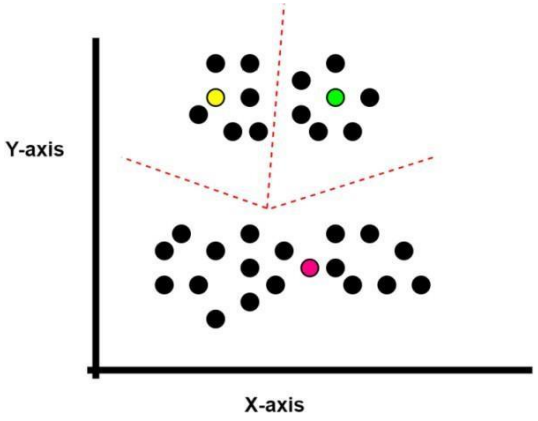


Fig 14. New set centroids in trial 2

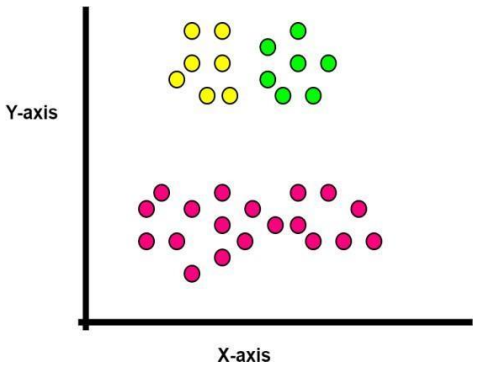


Fig 15. Formation of three different clusters in trial 2

4.3.1 Solution to the Problem:

To prevent random initialization, we have a solution called k-means++, which is an extension of k-means. We may also run the model more than twice. Because of poor random initialization, k-means frequently wind up in local optima [6].

k-means++ uses the following steps to avoid the problem of random initialization trap/

Step 1: select centroids at random for k clusters

Step 2: sum of squares distance between each point and each centroid

Step 3: for each data point in the collection, locate the shortest distance.

Step 4: To generate a new centroid, compute the mean for each cluster by finding out the data point number in each cluster.

4.4 SELECTING THE NUMBER OF CLUSTERS

Determining the correct number of clusters in a data set is a basic difficulty in partitioning clustering, such as k-means clustering, which requires the user to define the number of clusters k to be produced. The performance of the k-means clustering method is determined on the very efficient clusters it generates. Choosing the right amount of clusters, on the other hand, is a difficult task. There are numerous approaches for determining the best number of clusters, but we'll focus on the best approach for determining the count of clusters or k value. The appropriate number of clusters is rather subjective and is determined by the method for assessing similarity and the partitioning settings [7]. Some of the methods through which we can determine the number of clusters is as follow.

1. Direct methods: Direct techniques entail optimising a criterion, such as the sum of squares inside the confines of a cluster or the typical silhouette. The comparable procedures are known as the elbow and silhouette methods.
2. Statistical method: use null hypothesis evidence for comparison, a good example of statistical method is gap statistic.

Note: In addition to the elbow, silhouette, and gap statistic approaches, more than thirty other indices and methods for determining the appropriate number of clusters have been published.

4.4.1 Elbow method:

Most prominent methods for analysing the ideal number of clusters is the Elbow approach. WCSS value notion has been used in this approach [1]. Within Cluster Sum of Squares (WCSS) is a term that describes the total cluster variations held inside a cluster. The formula for calculating the WCSS value in the case of three clusters is shown in Equation 1:

$$WCSS = \sum X_{a \text{ in cluster } 1} \text{ distance } (X_a F_1)^2 + \sum X_{a \text{ in cluster } 2} \text{ distance } (X_a F_2)^2 + \sum X_{a \text{ in cluster } 3} \text{ distance } (X_a F_3)^2 \quad \text{(Equation 1)}$$

Referring to Equation1,

$\sum X_{a \text{ in cluster } 1}$ distance $(X_a F_1)^2$: It is an addition of the centroid and the square of the distance between every data point.

For calculating the distance between data points, any method among Manhattan or Euclidean distance can be used.

To determine the optimal value, the elbow method follows four steps which are as follows.

Step 1: It clusters a dataset using K-means for various K values. (ranges from 1-10).

Step 2: WCSS value will be computed for each value of K.

Step 3: Plots a line between estimated WCSS values and K clusters.

Step 4: The best K value is considered when a sharp point of bend or a plot point resembles an arm.

4.4.2 Silhouette Method

It assesses the clustering's quality. In other words, it establishes how well each object fits within its cluster. A high average silhouette width indicates good clustering. The silhouette method computes the average silhouette of observations for various values of k. The ideal number of clusters k is the one that maximises the average silhouette over a range of possible values for k. The silhouette method follows the below-mentioned steps to compute the optimal value [8].

Step 1: For the different values of k, a clustering algorithm will be computed.

Step 2: Compute the average silhouette for each value of k.

Step 3: Curve will be plotted based on several clusters.

Step 4: The optimal value will be considered based on the maximum location value.

4.4.3 Gap Statistic Method

The gap statistic compares the total intra-cluster variation for various values of k to their null reference distribution expected values. The best clusters will be calculated using the value that maximises the gap statistic (i.e that yields the largest gap statistic). This suggests that the clustering structure differs from a random uniform distribution of points in a substantial way. [8].

4.5 HIERARCHICAL CLUSTERING

In a hierarchical clustering procedure, the data is organised into a tree of groupings. In hierarchical clustering, each data point is regarded as a separate cluster. In data mining and statistics, hierarchical clustering (also known as hierarchical cluster analysis or HCA) is a cluster analysis method that seeks to establish a hierarchy of clusters [9].

Hierarchical clustering performs the following steps.

1. Identification of the two nearest together clusters.
2. Two maximum clusters get merged and these steps will continue until all the clusters get merged.

The goal of hierarchical clustering is to create a succession of nested clusters in a hierarchical order. The dendrogram is a type of diagram which represents hierarchical clustering. A Dendrogram is a tree-like graphic that counts the number of merges or splits in a series. The inverted tree graphically shows this hierarchy and describes the sequence in which factors are merged or clusters are broken up. In this algorithm, we will develop a hierarchy of clusters due to which this algorithm is known as hierarchical clustering [10].

There are two types of hierarchical clustering which have been explained in detail below.

4.5.1 Agglomerative Clustering:

Agglomerative clustering is the type of hierarchical clustering through which we can generate hierarchical clusters. It is a technique through which clusters can be developed hierarchically. In the initial stage of agglomerative clustering, each data point is considered to be its Cluster, and combine the cluster's nearest couples at each stage. (It's a bottom-up approach.) At initially, each data set is regarded as a separate entity or cluster. The clusters merge with other clusters in each cycle until only one cluster remains [11].

Agglomerative clustering consists of the following steps.

1. Computing the similarity of each cluster with other clusters or computing the proximity matrix.
2. Every data point is to be considered as a single and individual cluster.
3. Clusters that are nearest to one another and similar will get merged.
4. Re-compute the proximity matrix
5. Perform the above two steps again until a single cluster remains.

Now, let us understand this with a demonstration of an example, suppose we have seven data points named as 1,2,3,4,5,6,7.

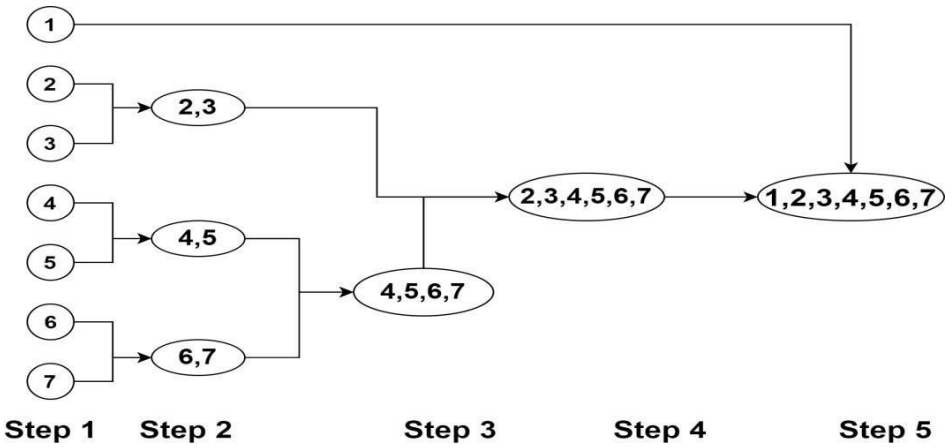


Fig 16. Agglomerative clustering

Referring to figure 16, we have seven different data points and considering them as n individual clusters, we have merged them in step 2 based on nearest neighbour and similarity. After merging in step 2, the proximity matrix has been computed again and merging of the clusters will continue until we get the final big cluster which is shown in figure 16 step 5.

4.5.2 Divisive Clustering

Divisive clustering is another type of hierarchical clustering which follows the bottom-up approach. This clustering technique is just the polar opposite of agglomerative clustering. In this, we consider all the data points as a single cluster and start separating them into single clusters or data points according to their incompatibility with other clusters [11]. It is just the opposite of the agglomerative clustering technique. Divisive clustering is explained with the help of diagrammatic representation of the above example shown in agglomerative clustering in figure 17.

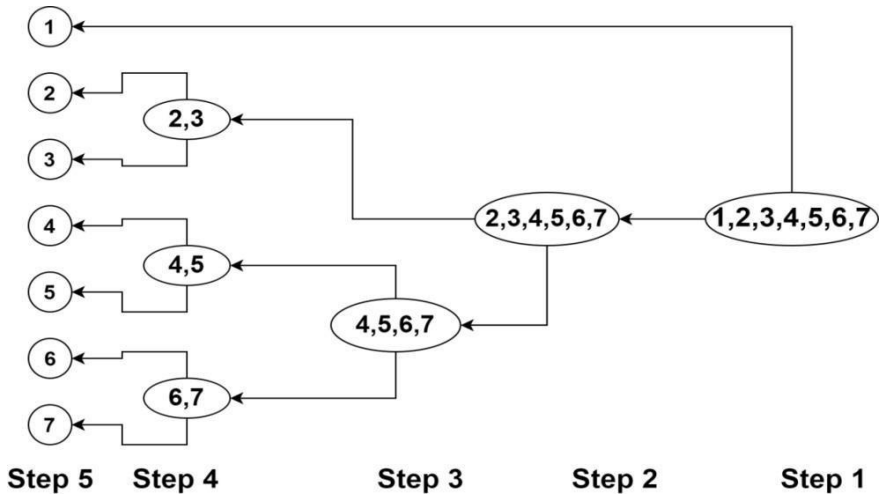


Fig 17. Divisive clustering

4.5.3 Dendrogram:

A dendrogram can be used to visualise the history of groups and determine the best number of clusters [12]. It involves the following points.

1. Determine the longest vertical distance between two clusters that do not intersect any of the others.
2. At both ends, draw a horizontal line.
3. The optimal number of clusters is equal to the number of vertical lines that pass through the horizontal line.

4.5.4 Criteria for Linkage:

A linkage criterion is referred to as the approach that how we are computing the distance between two clusters [13]. For calculating the distance between the clusters we have four different methods which are mentioned below.

1. **Single linkage:** It refers to the shortest distance between two different points in two different clusters.
2. **Complete linkage:** It refers to the longest distance between two data points of different clusters.
3. **Average linkage:** It is the average distance of a data point of one cluster to all the data points of another cluster.
4. **Ward linkage:** It is the addition of squared differences in all clusters.

4.5.5 Distance metric:

The distance metric is defined as the method by which we compute the distance between two different data points which affect the output of the algorithm [13]. There are two types of distance metrics we used for calculating the distance between the data points.

1. **Euclidean distance:** It refers to the shortest distance between two data points. For example, if c= (q, w) and g= (r, v) then the Euclidean distance will be $\sqrt{(q-r)^2 + (w-v)^2}$
2. **Manhattan distance:** It is defined as the sum of modulus difference between two data points. For example, if c= (q, w) and g= (r, v) then the manhattan distance will be $|q-r| + |w-v|$

4.5.6 Implementation of hierarchical clustering in python:

The implementation of hierarchical clustering in python involves some set of steps which are discussed as follows.

Step 1: Development of dataset:

As our coding part will be based on python therefore we need to create a database in a .csv file. The demo database has been shown in table 2 which demonstrate the employee's data of an organisation.

Table 2. Employees database

Index	EmpID	Gender	Age	Salary (In k rupees)	Work score
1	1	Male	20	25	10
2	2	Female	33	40	5
3	3	Male	21	25	6
4	4	Male	40	40	3
5	5	Female	28	25	7
6	6	Female	19	15	6
7	7	Male	26	25	9
8	8	Male	22	25	4
9	9	Male	35	40	9
10	10	Male	41	40	8

Using table 2 data's .csv file we will implement hierarchical clustering in python.

Step 2: Importing library

The very first step of implementing the algorithm in python is importing the necessary libraries so that we can able to use all the important inbuilt functions and classes which are needed to implement the clustering algorithm.

Code:

```
import pandas as p_do
import numpy as n_plo
from matplotlib import pyplot as p_df
from sklearn.cluster import AgglomerativeClustering
```

```
import scipy.cluster.hierarchy as s_iu
```

Step 3: Uploading and reading the dataset

After creating a .csv file, we need to upload and read that file into a database, so that at the time of implementation we can able to use it.

Code:

```
from google.colab import files
up = files.upload()
import io
f_r = p_do.read_csv(io.BytesIO(up['exc.csv']))
```

Step 4: Finding the optimal cluster number using a dendrogram

To find the optimal number of clusters, we use a dendrogram which gives the optimal number of clusters by showing the number of vertical lines which do not intersect the threshold value as shown in figure 18. In the below code ward linkage has been used for linking the data points of different clusters.

Code:

```
X = f_r.iloc[:, [3, 4]].values
den_gram = s_iu.dendrogram(s_iu.linkage(X, method='ward'))
```

Output:

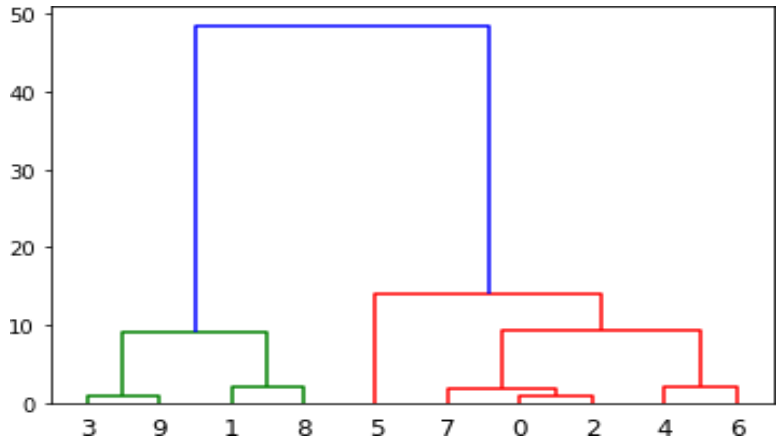


Fig 18. The output of step 4

Step 5: Determining the proximity of the cluster

In this step, AgglomerativeClustering instance has been created to estimate cluster proximity, find the euclidean distance, which measures the distance between points and ward linkage.

Code:

```
mod_1 = AgglomerativeClustering(n_clusters=5, affinity='euclidean', linkage='ward')
mod_1.fit(X)
labels = mod_1.labels_
```

print(labels)

Output:

[3 1 3 4 0 2 0 3 1 4]

Step 6: Displaying the number of output clusters

This is the last step of the implementation, which displays the different number of clusters and data points based on cluster proximity and distance between the data points. The generated clusters after the implementation of the algorithm has been shown in figure 19.

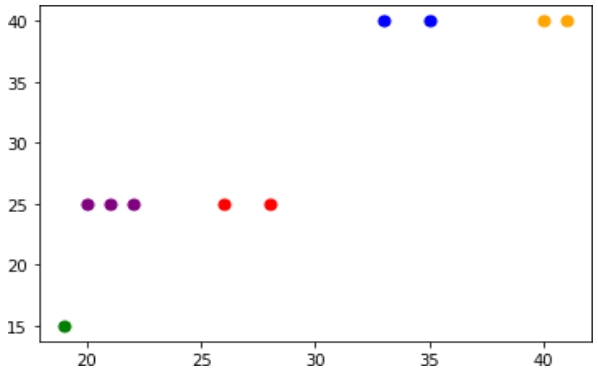


Fig 19. Generated clustering based on data point distance and cluster proximity

4.5.7 Advantages and disadvantages of hierarchical clustering:

Advantages:

1. There is no requirement for a specific number of clusters.
2. They may match relevant taxonomies.

Disadvantages:

1. It is impossible to reverse a choice to join two clusters once it has been taken.
2. O(n2 log(n)) is too slow for huge data collections.

4.5.8 Applications of hierarchical clustering:

1. Clustering of Senators from the United states through Twitter.
2. Used in chart based evolution studies
3. Virus tracking

4.6 SELF-CHECK QUESTIONS

Q1. An example of a movie recommendation system is?

A. Clustering
B. Classification
C. Regression
D. None of the above

Q2. Is it possible to use decision trees to perform clustering?

A. True
B. False

Q3. Given a minimal number of data points, which of the following is the most acceptable technique for data cleansing before performing clustering analysis:

1. Flavouring of variables
2. Outliers removal

A. Only 2
B. Only 1
C. 1 and 2
D. None of these

Q4. What is the bare minimum of variables required for clustering to work?

A. 1
B. 2
C. 3
D. 0

Q5. Is it reasonable to assume the same clustering results from two K-Mean clustering runs?

A. NO
B. YES

Q6. Is it conceivable that with K-Means, the assignment of observations to clusters does not alter over time?

A. NO
B. YES
C. Can’t say
D. None of these

Q7. Which of the following clustering techniques has difficulty with local optima convergence?

A. Agglomerative clustering
B. Divisive clustering
C. K-means clustering
D. Maximization clustering

Q8. Which algorithm is the most outlier-sensitive?

A. k-modes clustering
B. k-medoids clustering
C. k-medians clustering
D. k-mean clustering

Q9. In which of the following situations will K-Means clustering fail to produce satisfactory results?

1. With outliers

2. With different densities
3. Data points with round shapes
4. Data points with non-convex shapes

A. 2 and 3
B. 1 and 2
C. 1, 2 and 4
D. All of the above

Q10. Exploration should be the primary goal of hierarchical clustering.

A. True
B. False

Q11. Which of the following results from Hierarchical Clustering at the end?

A. final cluster centroids estimation
B. a tree that shows how close items are to one another
C. Each point is assigned to a cluster.
D. All of the mentioned above

Q12. Which of the following clustering approaches involves merging?

A. Hierarchical
B. Naïve bayes
C. Partitional
D. None of the above

Q13. Before using the K-Mean technique, it is necessary to scale the features. What is the explanation for this?

A. It will use the same weights for all features when calculating distance.
B. The identical clusters appear every time. If you employ feature scaling, or if you don't.
C. It is a significant stride in Manhattan distance, but not in Euclidian distance.
D. None of the above

Q14. Method for finding the optimal number of clusters in the k-mean clustering algorithm is?

A. Ecludian method
B. Manhattan method
C. Elbow method
D. None of the above

Q15. K-means is a non-deterministic algorithm that includes several rounds.

A. True
B. False

4.7 SUMMARY

This module will help students to understand various machine learning clustering algorithms along with their real-time applications, pros and cons. Along with this, the working and implementation of these clustering algorithms have been discussed in detail. In every section of the module, a basic introduction of the module followed by its working, assumptions taken for the algorithm to work, its pros and cons, a real-life application which covers the areas or fields where that particular algorithm has been used, and at last step by step implementation of the algorithms in a python programming language has been discussed in detail. The module also covers the vast diagrammatic representation of algorithm working and implementation, which will assist the student to understand the concept more clearly and deeply. In conclusion, this particular module will assist the student to understand the fundamental idea that how clustering can work in machine learning with the help of various machine learning clustering algorithms and their step by step work.

4.8 UNIT END QUESTIONS

Q1. What do you mean by k-means clustering?

Q2. Explain Hierarchical clustering in detail?

Q3. Explain the k-means random initialization trap in detail, with the help of an example?

Q4. State pros and cons of k-means clustering? Also, state its applications.

Q5. Which method should be used for finding an optimal number of the k-means cluster? Explain in detail.

Q6. Explain centroid point in the k-means algorithm?

Q7. State advantages, disadvantages and applications of hierarchical clustering

Q8. Explain the role of criteria linkage and distance metric in hierarchical clustering?

Q9. What is a dendrogram? Also, explain the elbow method along with its computational formula?

Q10. Implement k-means and hierarchical clustering with the help of python programming language?

REFERENCES

[1] https://www.javatpoint.com/k-means-clustering-algorithm-in-machine-learning

[2] https://towardsdatascience.com/understanding-k-means-clustering-in-machine-learning-6a6e67336aa1

[3] https://en.wikipedia.org/wiki/K-means_clustering

[4] https://www.geeksforgeeks.org/ml-random-intialization-trap-in-k-means/

[5] https://www.linkedin.com/pulse/everything-k-means-navya-rao/

[6] https://medium.datadriveninvestor.com/k-means-clustering-6f2dc458cce8

[7] https://www.geeksforgeeks.org/ml-determine-the-optimal-value-of-k-in-k-means-clustering/

[8] https://www.datanovia.com/en/lessons/determining-the-optimal-number-of-clusters-3-must-know-methods/

[9] https://www.geeksforgeeks.org/hierarchical-clustering-in-data-mining/

[10] https://towardsdatascience.com/understanding-the-concept-of-hierarchical-clustering-technique-c6e8243758ec

[11] https://www.analyticsvidhya.com/blog/2019/05/beginners-guide-hierarchical-clustering/

[12] https://en.wikipedia.org/wiki/Hierarchical_clustering

[13] https://www.displayr.com/what-is-hierarchical-clustering/#:~:text=Hierarchical%20clustering%2C%20also%20known%20as,broadly%20similar%20to%20each%20other.

MACHINE LEARNING
UNIT V: ARTIFICIAL NEURAL NETWORK

STRUCTURE

5.0 Objective

5.1 Introduction to ANN

5.2 Biology Neural Network

5.3 Artificial Neural Network Applications

5.4 Neural Network Architecture Type

5.5 Learning

5.6 Activation Function In Python

5.7 Summary

5.8 Practice Exercise

5.0 OBJECTIVE

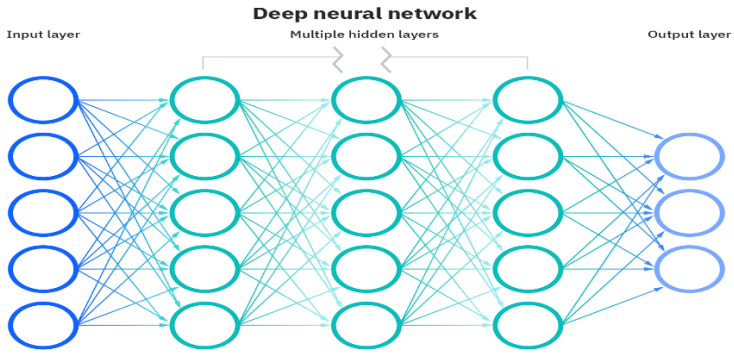
- Under standing biological neural networks; Usefulness and Applications of ANNs;
- Architectures of ANNs: Single layer, Multi layer, Competitive layer;
- Learning: Supervised and Unsupervised

5.1 INTRODUCTION TO ANN

Neural networks replicate the performance of the human brain, which allows computer programs to identify patterns and solve common issues in the fields of Machine learning, Deep learning, and Artificial Intelligence (AI).

Simulated Neural Networks (SNNs) or Artificial Neural Networks (ANNs), are also known as Neural Networks. They are at the core of Deep Learning and a subset of Machine learning algorithms. Their structure and name are motivated by the human brain

Artificial neural networks (ANNs) are composed of node layers, containing one or more hidden layers, an output layer, and an input layer. An artificial neuron or every node is attached to another and has an associated threshold and weight. A node is activated when the output of any single node is above the specific threshold value. An activated node can send data to the subsequent layer in the network. Else, no data can be passed to the subsequent layer in the network.



Neural networks depend on training data to study and progress their accuracy over time. Once the learning algorithms are ready with accuracy, then they become powerful tools in Artificial Intelligence and Computer Science, allow you to categorize and cluster data at a very high velocity. Works in image recognition or speech recognition can take few minutes concerning the manual identification done by human experts. Google‘s search algorithm is one of the famous neural networks.

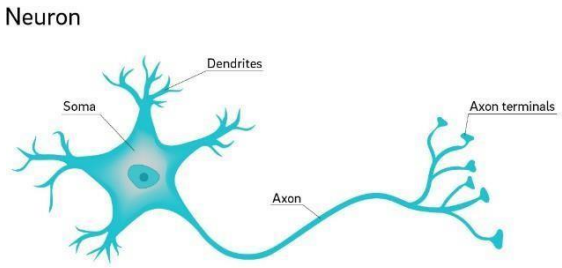
5.2 BIOLOGICAL NEURAL NETWORK

Researchers have tried many times to copy the biological systems, one of the results of such research is Artificial Neural Networks motivated by the living organism‘s biological neural

networks. On the other hand, they are very much dissimilar in many ways. For instance, the inspiration behind the invention of airplanes by researchers was birds, and similarly, the four-legged animals motivated researchers to develop cars.

The artificial parts are very powerful and make our life improved. The predecessors of artificial neurons are perceptrons, which were shaped to mimic certain parts of a biological neuron-like dendrite, axon, and cell body using electronics, mathematical models, and whatever partial data we have of biological neural networks.

5.2.1 Parts of a Biological Neural Network



In human beings, the brain is the control unit of the neural network. It has various subunits that take care of senses, vision, senses, hearing, and movement. The brain is connected to the rest of the body‘s sensors and actors through a dense network of nerves. There are almost 10^{11} neurons in the brain. These are the building blocks of the entire central nervous system of human beings.

The neuron is the basic building block of neural networks. The neuron is the fundamental unit of neural networks. A neuron is a cell, like any other cell in the body that contains a DNA code and is produced in the same way as other cells in biological systems. Even though each organism's DNA is different, the function is the same. The cell body (also known as Soma), axon and dendrites are the three major components of a neuron. Dendrites are like fibers that branch out in different directions and connect to a large number of cells in a cluster.

The axon receives signals from neighboring neurons and sends them to the other neurons through dendrites. A synapse connects the axon's terminating terminal to the dendrite. Axon is a long fiber that transfers the output signal along its length as electric impulses. Axons are the extensions of neurons. Each neuron has one axon. Axons act as a domino effect, passing impulses from one neuron to the next.

A typical human brain nerve cell is made up of four parts:

Function of Dendrite: Other neurons send signals to it.

Soma (cell body):

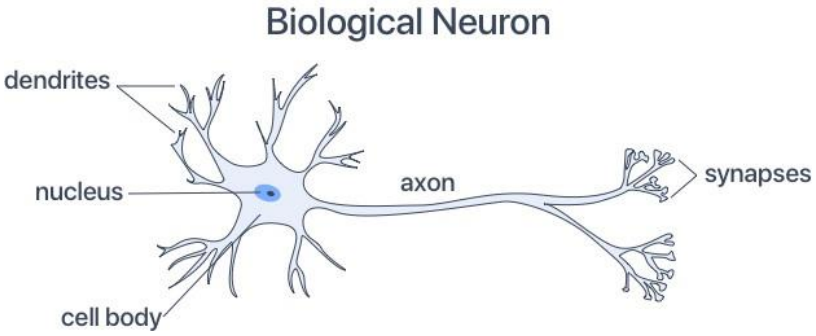
To generate input, it adds all of the incoming signals together.

Axon Structure:

When the neuron fires sum reaches a threshold value, and the signal goes down the axon to the other neurons.

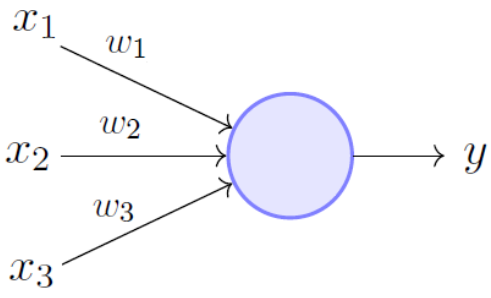
Synapses Working:

The strength (synaptic weights) of the synapses determine the quantity of signal sent.



The connections can be excitatory (increasing strength) ot inhibitory (decreasing strength) in nature. In general, a neural network consists of a connected network of billions of neurons with trillions of interconnections.

Since biological and artificial neural networks are so closely related, theoretical examination of biological neural networks is necessary for developing mathematical models for artificial neural networks. This new knowledge of the brain's neural networks has paved the way for the creation of artificial neural network systems and adaptive systems that can learn and adapt to new settings and inputs.



Perceptron Model (Minsky-Papert in 1969)

5.2.1 Working of BNN

Consider each node as a separate model, with weights, input data, an output, and a threshold. This is what the formula would look like:

$$\sum_{i=1}^{m} w_i x_i + bias = w_1x_1 + w_2x_2 + w_3x_3 + bias$$

∑i=1 mwixi + bias = w1x1 + w2x2 + w3x3 + bias

$$output = f(x) = \begin{cases} 1 \text{ if } \sum w_1x_1 + b \geq 0 \\ 0 \text{ if } \sum w_1x_1 + b < 0 \end{cases}$$

output = f(x) = 1 if ∑w1x1 + b>= 0; 0 if ∑w1x1 + b < 0

Weights are assigned once an input layer has been defined. These weights aid in determining the importance of any given variable, with larger ones contributing more to the output than smaller ones. After that, all of the inputs are multiplied by their respective weights and then added together. The output is then run through an activation function to determine the output. If the output reaches a certain threshold, the node "fires" (or activates), sending data to the network's next tier. As a result, one node's output becomes the input of the next node. The neural network is referred as a feed forward network since data is passed from one layer to the next.

We'll utilise supervised learning, or labelled datasets, to train the algorithm as we consider more practical use cases for neural networks, such as image recognition or classification. After training the model, its accuracy will be check using a loss or (cost) function. This is also known as the Mean Squared Error (MSE). In the equation below,

i represents the index of the sample,

y-hat is the predicted outcome,

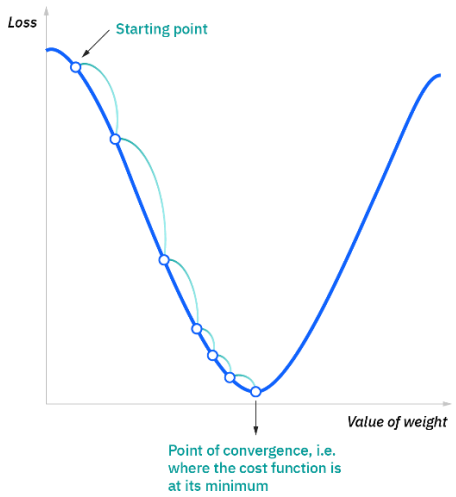
y is the actual value, and

m is the number of samples.

$$Cost\ Function = MSE = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y} - y)^2$$

The ultimate goal is to reduce our cost function in order to ensure that each given observation is correctly suited. The model employs the cost function and reinforcement learning to approach the local minimum or the point of convergence, as it adjusts its weights and bias. Gradient descent is the method by which the algorithm modifies its weights, allowing the model to

discover the best path to minimise the cost function (or minimise the mistakes). The parameters of the model adapt with each training case to progressively converge at the minimum.



The majority of deep neural networks are feed forward, which means they only flow one way, from input to output. Backpropagation, or moving in the reverse direction from output to input, is another way to train your model. Backpropagation allows us to calculate and attribute each neuron's error, allowing us to tweak and fit the model(s) parameters correctly.

5.3 ARTIFICIAL NEURAL NETWORK APPLICATIONS

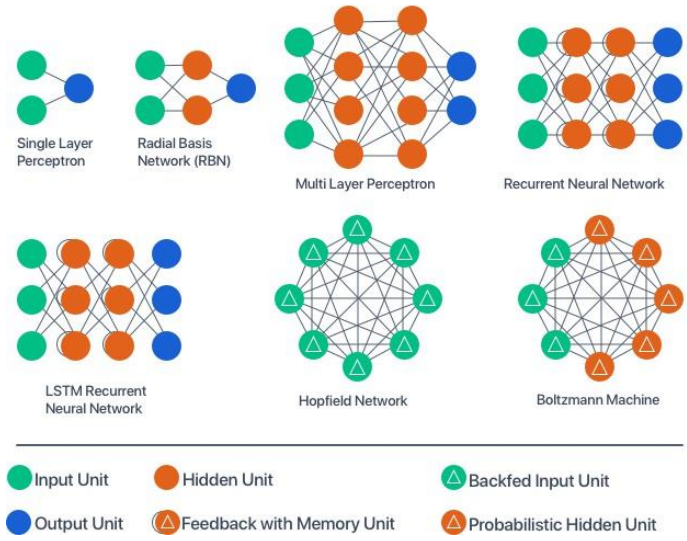
Image Compression – Neural networks receive and process large volumes of data all at once. As a result, they're useful for image compression. With the growth of the Internet and the increasing use of images on websites, using neural networks for image compression is a viable option.

Handwriting Recognition – The concept of handwriting recognition has gained a lot of traction. This is due to the growing popularity of handheld devices such as the Palm Pilot. As a result, neural networks can be used to recognise handwritten characters.

Stock Exchange Prediction – The stock market's day-to-day operations are extremely complex. Many factors influence whether a stock will rise or fall in value on any given day. As a result, neural networks can quickly review and sort a large amount of data. As a result, they can be used to forecast stock prices.

Traveling Salesman Problem The traveling salesman problem can also be solved using neural networks. However, this is simply an approximation to a certain degree.

5.4 NEURAL NETWORK ARCHITECTURE TYPE



5.4.1 Perceptron Model in Neural Networks

Neural Network is having two input units and one output unit with no hidden layers. These are also known as _single-layer perceptrons.'

5.4.2 Radial Basis Function Neural Network

These networks are similar to the feed-forward Neural Network, except radial basis function is used as these neurons' activation function.

5.4.3 Multilayer Perceptron Neural Network

These networks use more than one hidden layer of neurons, unlike single-layer perceptron. These are also known as Deep Feedforward Neural Networks.

5.4.4 Recurrent Neural Network

Type of Neural Network in which hidden layer neurons have self-connections. Recurrent Neural Networks possess memory. At any instance, the hidden layer neuron receives activation from the lower layer and its previous activation value.

5.4.5 Long Short-Term Memory Neural Network (LSTM)

The type of Neural Network in which memory cell is incorporated into hidden layer neurons is called LSTM network.

5.4.6 Hopfield Network

A fully interconnected network of neurons in which each neuron is connected to every other neuron. The network is trained with input patterns by setting a value of neurons to the desired pattern. Then its weights are computed. The weights are not changed. Once trained for one or more patterns, the network will converge to the learned patterns. It is different from other Neural Networks.

5.4.7 Boltzmann Machine Neural Network

These networks are similar to the Hopfield network, except some neurons are input, while others are hidden in nature. The weights are initialized randomly and learn through the backpropagation algorithm.

5.4.8 Modular Neural Network

It is the combined structure of different types of neural networks like multilayer perceptron, Hopfield Network, Recurrent Neural Network, etc., which are incorporated as a single module into the network to perform independent subtask of whole complete Neural Networks.

5.4.9 Physical Neural Network

In this type of Artificial Neural Network, electrically adjustable resistance material is used to emulate synapse instead of software simulations performed in the neural network.

5.5 LEARNING

5.5.1 Supervised Learning

Supervised learning, as the name indicates, has the presence of a supervisor as a teacher. Basically supervised learning is when we teach or train the machine using data that is well labeled. Which means some data is already tagged with the correct answer. After that, the machine is provided with a new set of examples(data) so that the supervised learning algorithm analyses the training data(set of training examples) and produces a correct outcome from labeled data.

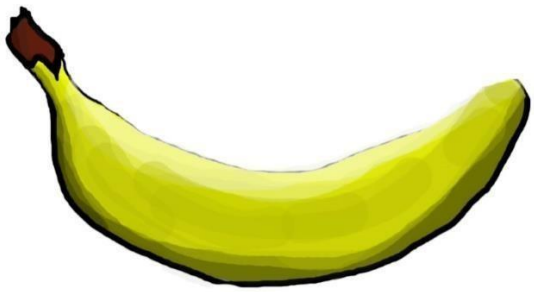
For instance, suppose you are given a basket filled with different kinds of fruits. Now the first step is to train the machine with all different fruits one by one like this:



If the shape of the object is rounded and has a depression at the top, is red in color, then it will be labeled as –Apple.

If the shape of the object is a long curving cylinder having Green-Yellow color, then it will be labeled as –Banana.

Now suppose after training the data, you have given a new separate fruit, say Banana from the basket, and asked to identify it.



Since the machine has already learned the things from previous data and this time has to use it wisely. It will first classify the fruit with its shape and color and would confirm the fruit name as BANANA and put it in the Banana category. Thus the machine learns the things from training data(basket containing fruits) and then applies the knowledge to test data(new fruit).

Supervised learning classified into two categories of algorithms:

Classification: A classification problem is when the output variable is a category, such as ―Red‖ or ―blue‖ or ―disease‖ and ―no disease‖.

Regression: A regression problem is when the output variable is a real value, such as ―dollars‖ or ―weight‖.

Supervised learning deals with or learns with ―labeled‖ data. This implies that some data is already tagged with the correct answer.

Types:-

Regression

Logistic Regression

Classification

Naive Bayes Classifiers

K-NN (k nearest neighbors)

Decision Trees

Support Vector Machine

Advantages:-

Supervised learning allows collecting data and produces data output from previous experiences.

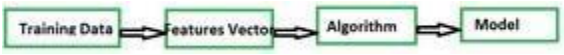
Helps to optimize performance criteria with the help of experience.

Supervised machine learning helps to solve various types of real-world computation problems.

Disadvantages:-

Classifying big data can be challenging.

Training for supervised learning needs a lot of computation time. So, it requires a lot of time.



Supervised learning is a machine learning task where an algorithm is trained to find patterns using a dataset. The supervised learning algorithm uses this training to make input-output inferences on future datasets. In the same way a teacher (supervisor) would give a student homework to learn and grow knowledge, supervised learning gives algorithms datasets so it too can learn and make inferences.

To illustrate how supervised learning works, let‘s consider an example of predicting the marks of a student based on the number of hours he studied.

Mathematically,

$Y = f(X) + C$

is broken down as follows:

f will be the relation between the marks and number of hours the student prepared for an exam.

X is the INPUT (Number of hours he prepared).

Y is the output (Marks the student scored in the exam).

C will be a random error.

The ultimate goal of the supervised learning algorithm is to predict *Y* with the maximum accuracy for a given new input *X*. There are several ways to implement supervised learning and we‘ll explore some of the most commonly used approaches.

Based on the given data sets, the machine learning problem is categorized into two types: **classification** and **regression**. If the given data has both input (training) values and output (target) values, then it is a classification problem. If the dataset has continuous numerical values of attributes without any target labels, then it is a regression problem.

Below is an example of where you can use supervised learning and unsupervised learning.

Classification: Has the output label. Is it a Cat or Dog?

Regression: How much will the house sell for?

Classification

Consider the example of a medical researcher who wants to analyze breast cancer data to predict one of three specific treatments a patient should receive. This data analysis task is called classification, and a model or classifier is constructed to predict class labels, such as ―treatment A,‖ ―treatment B‖ or ―treatment C.‖

Classification is a prediction problem that predicts the categorical class labels, which are discrete and unordered. It is a two-step process, consisting of a learning step and a classification step.

METHODS IN CLASSIFICATION AND CHOOSING THE BEST.

There are several classification techniques that one can choose based on the type of dataset they're dealing with. Below is a list of a few widely used traditional classification techniques:

K—nearest neighbor

Decision trees

Naïve Bayes

Support vector machines

In the first step, the classification model builds the classifier by analyzing the training set. Next, the class labels for the given data are predicted. The dataset tuples and their associated class labels under analysis are split into a training set and test set. The individual tuples that make up the training set are randomly sampled from the dataset under analysis. The remaining tuples form the test set and are independent of the training tuples, meaning they will not be used to build the classifier.

The test set is used to estimate the predictive accuracy of a classifier. The accuracy of a classifier is the percentage of test tuples that are correctly classified by the classifier. To achieve higher accuracy, the best way is to test out different algorithms and try different parameters within each algorithm. The best one can be selected by cross-validation.

To choose a good algorithm for a problem, parameters such as accuracy, training time, linearity, number of parameters and special cases must be taken into consideration for different algorithms.

Implementing KNN In Scikit-Learn On Iris Dataset

The first step in applying our machine learning algorithm is to understand and explore the given dataset. In this example, we'll use the Iris dataset imported from the scikit-learn package. Now let‘s dive into the code and explore the IRIS dataset.

Before getting started, make sure you install the following python packages using pip.

pip install pandas

pip install matplotlib

pip install scikit-learn

In this snippet of code, we learn about the attributes of the IRIS dataset using a few methods in pandas. (eda_iris_dataset.py on GitHuB)

fromsklearn import datasets

import pandas as pd

importmatplotlib.pyplot as plt

Loading IRIS dataset from scikit-learn object into iris variable.

iris = datasets.load_iris()

Prints the type/type object of iris

print(type(iris))

<class 'sklearn.datasets.base.Bunch'>

prints the dictionary keys of iris data

print(iris.keys())

prints the type/type object of given attributes

print(type(iris.data), type(iris.target))

prints the no of rows and columns in the dataset

print(iris.data.shape)

```
# prints the target set of the data
print(iris.target_names)


# Load iris training dataset
X = iris.data
# Load iris target set
Y = iris.target
# Convert datasets' type into dataframe
df = pd.DataFrame(X, columns=iris.feature_names)
# Print the first five tuples of dataframe.
print(df.head())
```

Output:

<class 'sklearn.datasets.base.Bunch'>

dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names'])]

<class 'numpy.ndarray'><class 'numpy.ndarray'>

(150, 4)

['setosa' 'versicolor' 'virginica']

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	**5.1**	**3.5**	**1.4**	**0.2**
1	**4.9**	**3.0**	**1.4**	**0.2**
2	**4.7**	**3.2**	**1.3**	**0.2**
3	**4.6**	**3.1**	**1.5**	**0.2**
4	**5.0**	**3.6**	**1.4**	**0.2**

K-nearest neighbors in Scikit-learn.

An algorithm is said to be a **lazy learner** if it simply stores the tuples of the training set and waits until the test tuple is given. Only when it sees the test tuple does it perform generalization to classify the tuple based on its similarity to the stored training tuples.

K-nearest neighbor (k-NN) classifier is a lazy learner.

Based on learning by analogy, k-NN compares a given test tuple with training tuples that are similar to it. The training tuples are described by *n* attributes. Each tuple represents a point in an *n*-dimensional space. In this way, all training tuples are stored in *n*-dimensional pattern space. When given an unknown tuple, a k-NN classifier searches the pattern space for the k training tuples that are closest to the unknown tuple. These k training tuples are the k ―nearest neighbors‖ of the unknown tuple.

―Closeness‖ is defined regarding a distance metric, such as Euclidean distance. A good value for K is determined experimentally.

In this snippet, we give import k-NN classifier from sklearn and apply to our input data which then classifies the flowers. (http://knn_iris_dataset.py on GitHub)

```
fromsklearn import datasets

fromsklearn.neighbors import KNeighborsClassifier


# Load iris dataset from sklearn

iris = datasets.load_iris()


# Declare an of the KNN classifier class with the value with neighbors.

knn = KNeighborsClassifier(n_neighbors=6)


# Fit the model with training data and target values

knn.fit(iris['data'], iris['target'])

# Provide data whose class labels are to be predicted

X = [

   [5.9, 1.0, 5.1, 1.8],
```

```
    [3.4, 2.0, 1.1, 4.8],
]

# Prints the data provided
print(X)

# Store predicted class labels of X
prediction = knn.predict(X)

# Prints the predicted class labels of X
print(prediction)
```

Output:

[1 1]

Here,

0 corresponds versicolor

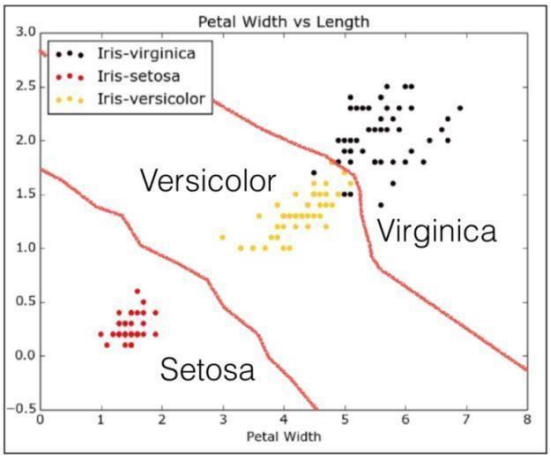
1 corresponds virginica

2 corresponds setosa

Based on the given input, the machine predicted the both flowers are versicolor using k-NN.

K-Nn Intuition For Iris Dataset Classification

If we plot the classified data using the k-NN algorithm in the Iris dataset this is how the flowers are categorized based on the features: the x-axis represents the petal width and the y-axis represents the petal length.



5.5.2 Unsupervised learning

Unsupervised learning is the training of a machine using information that is neither classified nor labeled and allowing the algorithm to act on that information without guidance. Here the task of the machine is to group unsorted information according to similarities, patterns, and differences without any prior training of data.

Unlike supervised learning, no teacher is provided that means no training will be given to the machine. Therefore the machine is restricted to find the hidden structure in unlabeled data by itself.
For instance, suppose it is given an image having both dogs and cats which it has never seen.



Thus the machine has no idea about the features of dogs and cats so we can‘t categorize it as _dogs and cats _. But it can categorize them according to their similarities, patterns, and differences, i.e., we can easily categorize the above picture into two parts. The first may contain all pics having dogs in it and the second part may contain all pics having cats in it. Here you didn‘t learn anything before, which means no training data or examples.

It allows the model to work on its own to discover patterns and information that was previously undetected. It mainly deals with unlabelled data.

Unsupervised learning is classified into two categories of algorithms:

Clustering: A clustering problem is where you want to discover the inherent groupings in the data, such as grouping customers by purchasing behavior.

Association: An association rule learning problem is where you want to discover rules that describe large portions of your data, such as people that buy X also tend to buy Y.

Types of Unsupervised Learning:-

- Clustering
- Exclusive (partitioning)
- Agglomerative
- Overlapping
- Probabilistic
- Clustering Types:-
    - Hierarchical clustering
    - K-means clustering
    - Principal Component Analysis
    - Singular Value Decomposition
    - Independent Component Analysis

Supervised vs. Unsupervised Machine Learning

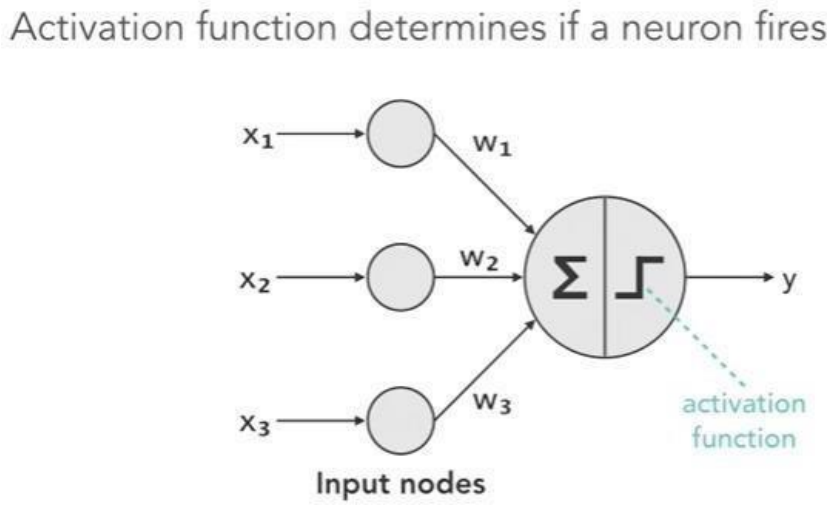
Parameters	Supervised machine learning	Unsupervised machine learning
Input Data	Algorithms are trained using labeled data.	Algorithms are used against data that is not labeled
Computational Complexity	Simpler method	Computationally complex
Accuracy	Highly accurate	Less accurate

In supervised learning, we start by importing a dataset containing training attributes and the target attributes. The supervised learning algorithm will learn the relation between training examples and their associated target variables, then apply that learned relationship to classify entirely new inputs (without targets).

5.6 ACTIVATION FUNCTION IN PYTHON

fromIPython.displayimportImage

Image(filename='data/Activate_functions.png')



Out[2]:

5.6.1 Binary Step Activation Function

Binary step function returns value either 0 or 1.

It returns '0' if the input is the less then zero

It returns '1' if the input is greater than zero

In [3]:

defbinaryStep(x):

''' It returns '0' is the input is less then zero otherwise it returns one '''

returnnp.heaviside(x,1)

In [4]:

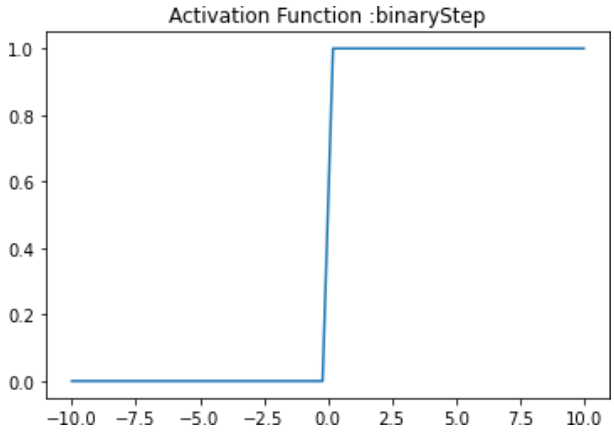
x=np.linspace(-10,10)

```
plt.plot(x,binaryStep(x))

plt.axis('tight')

plt.title('Activation Function :binaryStep')

plt.show()
```



5.6.2 Linear Activation Function

Linear functions are pretty simple. It returns what it gets as input.

In [5]:

```
deflinear(x):

''' y = f(x) It returns the input as it is'''

returnx
```

In [6]:

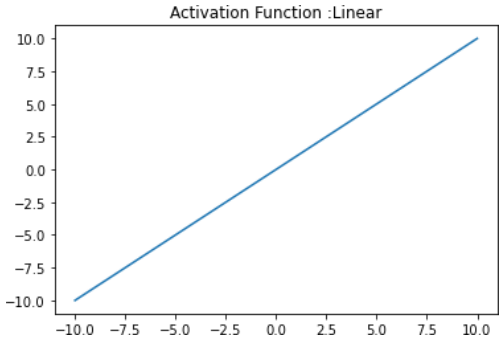
```
x=np.linspace(-10,10)

plt.plot(x,linear(x))

plt.axis('tight')

plt.title('Activation Function :Linear')

plt.show()
```



5.6.3 Sigmoid Activation Function

Sigmoid function returns the value beteen 0 and 1. For activation function in deep learning network, Sigmoid function is considered not good since near the boundaries the network doesn't learn quickly. This is because gradient is almost zero near the boundaries.

In [7]:

defsigmoid(x):

''' It returns 1/(1+exp(-x)). where the values lies between zero and one '''

return1/(1+np.exp(-x))

In [8]:

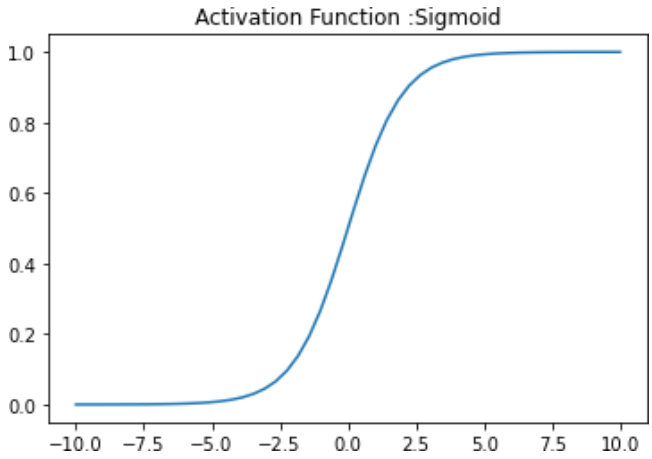
x=np.linspace(-10,10)

plt.plot(x,sigmoid(x))

plt.axis('tight')

plt.title('Activation Function :Sigmoid')

plt.show()



5.6.4 Tanh Activation Function

Tanh is another nonlinear activation function. Tanh outputs between -1 and 1. Tanh also suffers from gradient problem near the boundaries just as Sigmoid activation function does.

In [9]:

deftanh(x):

''' It returns the value (1-exp(-2x))/(1+exp(-2x)) and the value returned will be lies in between -1 to 1.'''

returnnp.tanh(x)

In [10]:

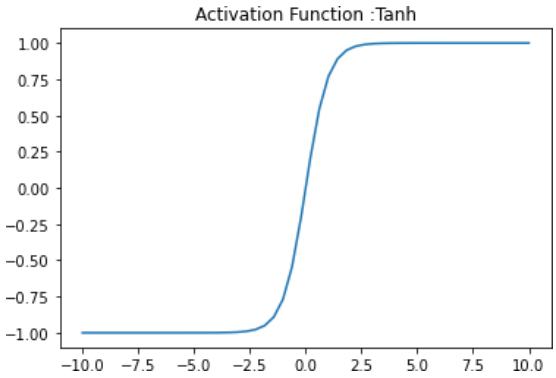
x=np.linspace(-10,10)

plt.plot(x,tanh(x))

plt.axis('tight')

plt.title('Activation Function :Tanh')

plt.show()



5.6.5 RELU Activation Function

RELU is more well known activation function which is used in the deep learning networks. RELU is less computational expensive than the other non linear activation functions.

RELU returns 0 if the x (input) is less than 0

RELU returns x if the x (input) is greater than 0

In [11]:

defRELU(x):

''' It returns zero if the input is less than zero otherwise it returns the given input. '''

x1=[]

foriinx:

ifi<0:

x1.append(0)

else:

x1.append(i)

returnx1

In [12]:

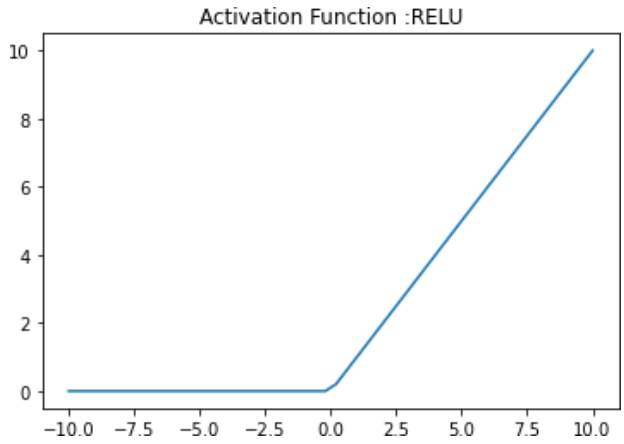
x=np.linspace(-10,10)

plt.plot(x,RELU(x))

plt.axis('tight')

plt.title('Activation Function :RELU')

plt.show()



5.6.6 Softmax Activation Function

Softmax turns logits, the numeric output of the last linear layer of a multi-class classification neural network into probabilities.

We can implement the Softmax function in Python as shown below.

In [13]:

defsoftmax(x):

''' Compute softmax values for each sets of scores in x. '''

returnnp.exp(x)/np.sum(np.exp(x),axis=0)

In [14]:

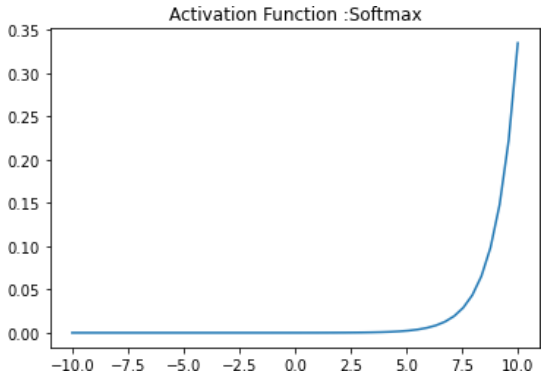
x=np.linspace(-10,10)

plt.plot(x,softmax(x))

plt.axis('tight')

plt.title('Activation Function :Softmax')

plt.show()



his is an example plot from the tutorial which accompanies an explanation of the support vector machine GUI.

importnumpyasnp

frommatplotlibimportpyplotasplt

fromsklearnimportsvm

data that is linearly separable

deflinear_model(rseed=42, n_samples=30):

" Generate data according to a linear model"

np.random.seed(rseed)

data=np.random.normal(0, 10, (n_samples, 2))

data[:n_samples//2] -=15

data[n_samples//2:] +=15

labels=np.ones(n_samples)

labels[:n_samples//2] =-1

return data, labels

X, y =linear_model()

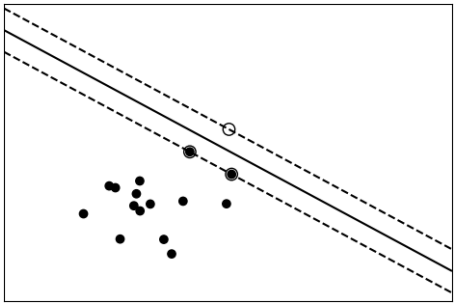
clf=svm.SVC(kernel='linear')

clf.fit(X, y)

```
plt.figure(figsize=(6, 4))
ax=plt.subplot(111, xticks=[], yticks=[])
ax.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.bone)
ax.scatter(clf.support_vectors_[:, 0],
clf.support_vectors_[:, 1],
        s=80, edgecolors="k", facecolors="none")

delta=1
y_min, y_max=-50, 50
x_min, x_max=-50, 50
x =np.arange(x_min, x_max+ delta, delta)
y =np.arange(y_min, y_max+ delta, delta)
X1, X2 =np.meshgrid(x, y)
Z =clf.decision_function(np.c_[X1.ravel(), X2.ravel()])
Z =Z.reshape(X1.shape)

ax.contour(X1, X2, Z, [-1.0, 0.0, 1.0], colors='k',
linestyles=['dashed', 'solid', 'dashed'])
```



data with a non-linear separation

```
defnonlinear_model(rseed=42, n_samples=30):
radius=40*np.random.random(n_samples)
far_pts= radius >20
radius[far_pts] *=1.2
radius[~far_pts] *=1.1

theta=np.random.random(n_samples) *np.pi*2

data=np.empty((n_samples, 2))
data[:, 0] = radius *np.cos(theta)
data[:, 1] = radius *np.sin(theta)

labels=np.ones(n_samples)
labels[far_pts] =-1

return data, labels
X, y =nonlinear_model()
clf=svm.SVC(kernel='rbf', gamma=0.001, coef0=0, degree=3)
clf.fit(X, y)

plt.figure(figsize=(6, 4))
ax=plt.subplot(1, 1, 1, xticks=[], yticks=[])
ax.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.bone, zorder=2)

ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
```

```
        s=80, edgecolors="k", facecolors="none")


delta=1
y_min, y_max=-50, 50
x_min, x_max=-50, 50
x =np.arange(x_min, x_max+ delta, delta)
y =np.arange(y_min, y_max+ delta, delta)
X1, X2 =np.meshgrid(x, y)
Z =clf.decision_function(np.c_[X1.ravel(), X2.ravel()])
Z =Z.reshape(X1.shape)
ax.contour(X1, X2, Z, [-1.0, 0.0, 1.0], colors='k',
linestyles=['dashed', 'solid', 'dashed'], zorder=1)
plt.show()
```

MACHINE LEARNING
UNIT VI: SUPERVISED MODELS

STRUCTURE

6.0 Objectives

6.1 Hebb Net: Algorithm, Application For And Problem

- **6.1.1 Hebbian Learning Rule Algorithm**
- **6.1.2 Implementing AND Gate**

6.2 Perceptron

- **6.2.1 Perceptron Learning Rate**
- **6.2.2 Perceptron Function**
- **6.2.3 Inputs of a Perceptron**
- **6.2.4 Error in Perceptron**

6.3 ADALINE

- **6.3.1 Architecture**
- **6.3.2 Training Algorithm**
- **6.3.3 Testing Algorithm**
- **6.3.4 Adaline python implementation**

6.4 Conclusions

6.5 Practice Exercises

6.0 OBJECTIVE

Supervised learning is the sorts of AI wherein machines are trained utilizing well "marked" preparing information, and on premise of that information, machines foresee the output. The labeled data implies some information is now labeled with the right output .

The labeled training data work as a supervisor to the mchine which trains the machine to predict correctly in supervised learning.It is similar to te concept that teacher teaches the student to learn the concepts better.

In reality, supervised learning can be utilized for Risk Assessment, Image arrangement, Fraud Detection, spam sifting, and so on

6.1 HEBB NET: ALGORITHM, APPLICATION FOR AND PROBLEM

The Hebbian rule was the principal learning rule. In 1949 Donald Hebb created it as learning calculation of the unsupervised neural network. We can utilize it to distinguish how to improve node weights of a network.

The hebb learning rule is assumed to be-the weight associated with two neurons increases if 2 neurons activated and deactivated simultaneously. For neurons working in the contrary phase, the weight between them should diminish.The wieght should be constant if no signal correlation is there.

The Hebbian learning rule describes the formula as follows:

$$W_{ij} = x_i * x_j$$

6.1.1 Hebbian Learning Rule Algorithm :

1. Intialise weights to zero, $w_i = 0$ for i=1 to n, and bias to zero.
2. For each input vector I: t(target output pair), repeat steps 3-5.
3. Set activations for input units with the I as $X_i = S_i$ for i = 1 to n.
4. Set the corresponding output value to the output neuron, i.e. y = t.
5. Update weight and bias by applying Hebb rule for all i = 1 to n:

$$w_i(new) = w_i(old) + x_i y$$

$$b(new) = b(old) + y$$

6.1.2 Implementing AND Gate

INPUT				TARGET	
	x_1	x_2	b		y
X_1	-1	-1	1	Y_1	-1
X_2	-1	1	1	Y_2	-1
X_3	1	-1	1	Y_3	-1
X_4	1	1	1	Y_4	1

Truth Table of AND Gate using bipolar sigmoidal function

There are 4 training samples, so there will be 4 iterations. Also, the activation function used here is Bipolar Sigmoidal Function so the range is [-1,1].

Step 1 :
Set weight and bias to zero, w = [0 0 0]^T and b = 0.
Step 2 :
Set input vector $X_i = S_i$ for i = 1 to 4.
X_1 = [-1 -1 1]^T
X_2 = [-1 1 1]^T
X_3 = [1 -1 1]^T

X_4 = [1 1 1]^T
Step 3 :
Output value is set to y = t.

Step 4 :
Modifying weights using Hebbian Rule:

First iteration –

w(new) = w(old) + x_1y_1 = [0 0 0]^T + [-1 -1 1]^T . [-1] = [1 1 -1]^T
For the second iteration, the final weight of the first one will be used and so on.

Second iteration –

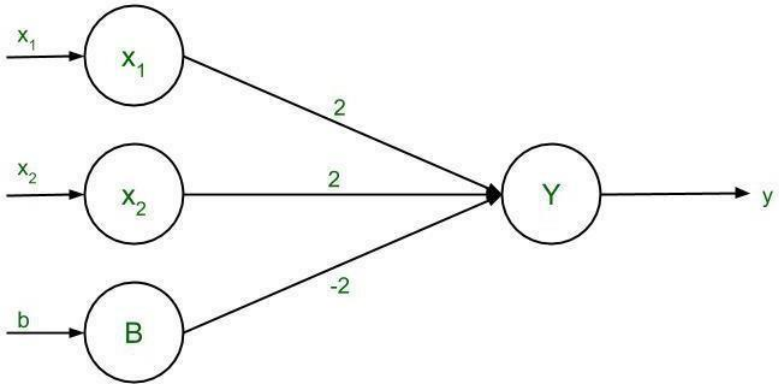
w(new) = [1 1 -1]^T + [-1 1 1]^T . [-1] = [2 0 -2]^T
Third iteration –

w(new) = [2 0 -2]^T + [1 -1 1]^T . [-1] = [1 1 -3]^T
Fourth iteration –

w(new) = [1 1 -3]^T + [1 1 1]^T . [1] = [2 2 -2]^T

So, the final weight matrix is [2 2 -2]^T

Testing the network :

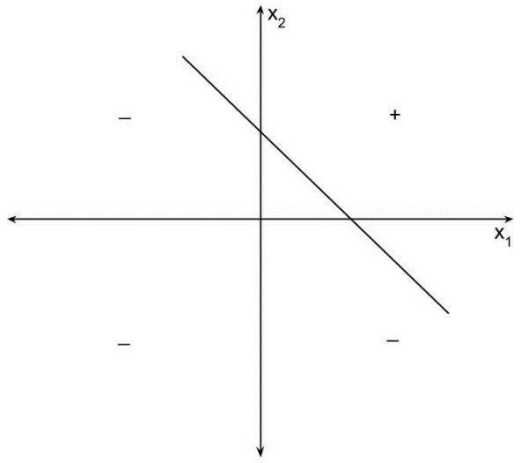


The network with the final weights

For $x_1 = -1$, $x_2 = -1$, b = 1, Y = (-1)(2) + (-1)(2) + (1)(-2) = -6
For $x_1 = -1$, $x_2 = 1$, b = 1, Y = (-1)(2) + (1)(2) + (1)(-2) = -2
For $x_1 = 1$, $x_2 = -1$, b = 1, Y = (1)(2) + (-1)(2) + (1)(-2) = -2
For $x_1 = 1$, $x_2 = 1$, b = 1, Y = (1)(2) + (1)(2) + (1)(-2) = 2
The results are all compatible with the original table.

Decision Boundary :

$2x_1 + 2x_2 – 2b = y$
Replacing y with 0, $2x_1 + 2x_2 – 2b = 0$
Since bias, b = 1, so $2x_1 + 2x_2 – 2(1) = 0$
$2(x_1 + x_2) = 2$
The final equation, $x_2 = -x_1 + 1$



Decision Boundary of AND Function

Implementation of Hebb Net

#Learning Rules #

```
import math

def computeNet(input, weights):
    net = 0
    for i in range(len(input)):
        net = net + input[i]*weights[i]
        print "NET:"
        print net
    return net

def computeFNetBinary(net):
    f_net = 0
    if(net>0):
        f_net = 1
    if(net<0):
        f_net = -1
    return f_net

def computeFNetCont(net):
    f_net = 0
    f_net = (2/(1+math.exp(-net)))-1
    return f_net

def hebb(f_net):
    return f_net
```

```
def perceptron(desired, actual):
        return (desired-actual)

def widrow(desired, actual):
        return (desired-actual)

def adjustWeights(inputs, weights, last, binary, desired, rule):
    c = 1
    if(last):
         print "COMPLETE"
                return
    current_input = inputs[0]
    inputs = inputs[1:]
        if desired :
                current_desired = desired[0]
                desired = desired[1:]
    if len(inputs) == 0:
         last = True
    net = computeNet(current_input, weights)
    if(binary):
         f_net = computeFNetBinary(net)
    else:
         f_net = computeFNetCont(net)
        if rule == "hebb":
        r = hebb(f_net)
        elif rule == "perceptron":
                r = perceptron(current_desired, f_net)
        elif rule == "widrow":
                r = widrow(current_desired, net)
    del_weights = []
```

```
        for i in range(len(current_input)):
            x = (c*r)*current_input[i]
            del_weights.append(x)
            weights[i] = x
        print("NEW WEIGHTS:")
        print(weights)
        adjustWeights(inputs, weights, last, binary, desired, rule)

if __name__=="__main__":
        #total_inputs = (int)raw_input("Enter Total Number of Inputs)
        #vector_length = (int)raw_input("Enter Length of vector)
        total_inputs = 3
        vector_length = 4
        #for i in range(vector_length):
        #weight.append(raw_input("Enter Initial Weight:")
        weights = [1,-1,0,0.5]
        inputs = [[1,-2,1.5,0],[1,-0.5,-2,-1.5],[0,1,-1,1.5]]
            desired = [1,2,1,-1]
        print("BINARY HEBB!")
        adjustWeights(inputs, [1,-1,0,0.5], False, True, None, "hebb")
        print("CONTINUOUS HEBB!")
        adjustWeights(inputs, [1,-1,0,0.5], False, False, None, "hebb")
            print("PERCEPTRON!")
            adjustWeights(inputs, [1,-1,0,0.5], False, True, desired, "perceptron")
            print("WIDROW HOFF!")
            adjustWeights(inputs, [1,-1,0,0.5], False, True, desired, "widrow")
BINARY HEBB!
NET:
3.0
NEW WEIGHTS:
```

[1, -2, 1.5, 0]

NET:

-1.0

NEW WEIGHTS:

[-1, 0.5, 2, 1.5]

NET:

0.75

NEW WEIGHTS:

[0, 1, -1, 1.5]

COMPLETE

CONTINUOUS HEBB!

NET:

3.0

NEW WEIGHTS:

[0.90514825364486673, -1.8102965072897335, 1.3577223804673002, 0.0]

NET:

-0.905148253645

NEW WEIGHTS:

[-0.42401264054072996, 0.21200632027036498, 0.84802528108145991, 0.63601896081109488]

NET:

0.318009480406

NEW WEIGHTS:

[0.0, 0.15767814164392502, -0.15767814164392502, 0.23651721246588753]

COMPLETE

PERCEPTRON!

NET:

3.0

NEW WEIGHTS:

[0, 0, 0.0, 0]

NET:

0.0

NEW WEIGHTS:

[2, -1.0, -4, -3.0]

NET:

-1.5

NEW WEIGHTS:

[0, 2, -2, 3.0]

COMPLETE

WIDROW HOFF!

NET:

3.0

NEW WEIGHTS:

[-2.0, 4.0, -3.0, -0.0]

NET:

2.0

NEW WEIGHTS:

[0.0, -0.0, -0.0, -0.0]

NET:

0.0

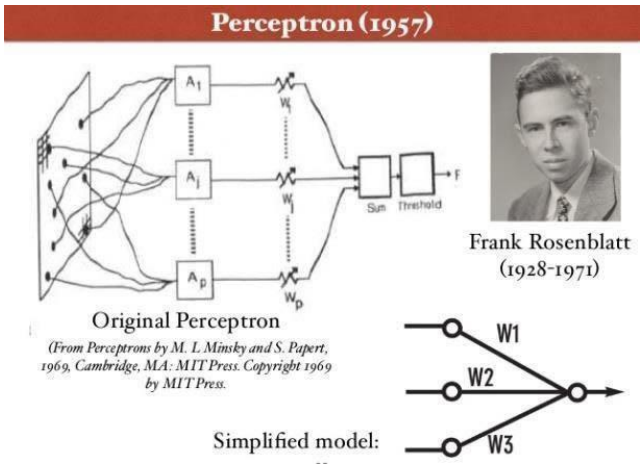
NEW WEIGHTS:

[0.0, 1.0, -1.0, 1.5]

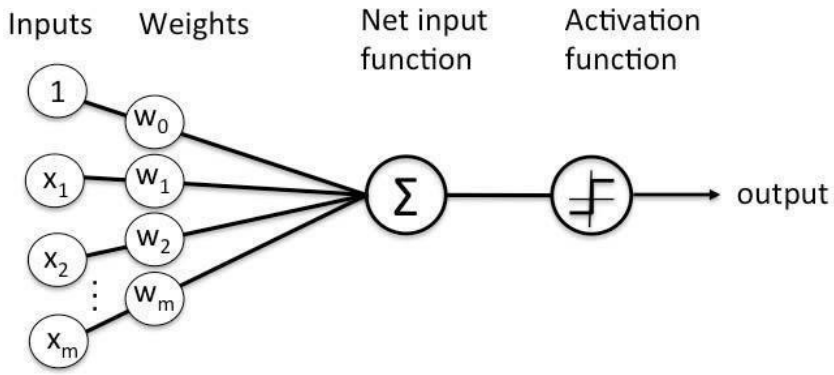
COMPLETE

6.2 PERCEPTRON

Perceptron is a neural network unit (an artificial neuron) that does certain computations to detect features or business intelligence in the input data. And this will give you an in-depth knowledge of Perceptron and its activation functions.



Perceptron was introduced by Frank Rosenblatt in 1957. He proposed a Perceptron learning rule based on the original MCP neuron. A Perceptron is an algorithm for supervised learning of binary classifiers. This algorithm enables neurons to learn and processes elements in the training set one at a time.



There are two types of Perceptrons: Single layer and Multilayer.

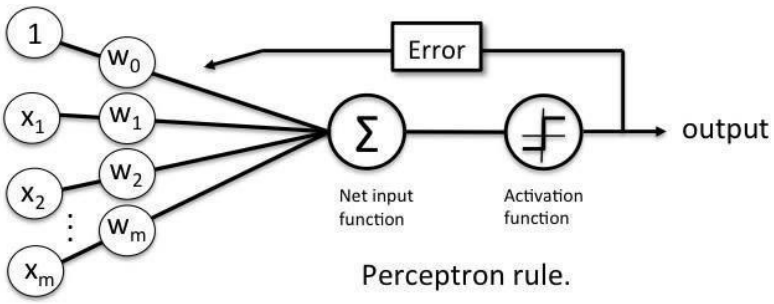
- Single layer - Single layer perceptrons can learn only linearly separable patterns
- Multilayer - Multilayer perceptrons or feedforward neural networks with two or more layers have the greater processing power

The Perceptron algorithm learns the weights for the input signals to draw a linear decision boundary.

This enables you to distinguish between the two linearly separable classes +1 and -1.

6.2.1 Perceptron Learning Rule

Perceptron Learning Rule states that the algorithm would automatically learn the optimal weight coefficients. The input features are then multiplied with these weights to determine if a neuron fires or not.



Perceptron rule.

The Perceptron receives multiple input signals, and if the sum of the input signals exceeds a certain threshold, it either outputs a signal or does not return an output. In the context of supervised learning and classification, this can then be used to predict the class of a sample.

6.2.2 Perceptron Function

Perceptron is a function that maps its input "x," which is multiplied with the learned weight coefficient; an output value "f(x)"is generated.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

In the equation given above:

- "w" = vector of real-valued weights
- "b" = bias (an element that adjusts the boundary away from origin without any dependence on the input value)
- "x" = vector of input x values

$$\sum_{i=1}^{m} w_i x_i$$

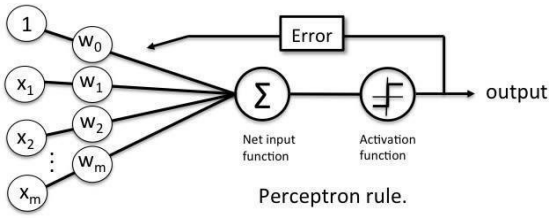
- "m" = number of inputs to the Perceptron

The output can be represented as “1” or “0.” It can also be represented as “1” or “-1” depending on which activation function is used.

Let us learn the inputs of a perceptron in the next section.

6.2.3 Inputs of a Perceptron

A Perceptron accepts inputs, moderates them with certain weight values, then applies the transformation function to output the final result. The image below shows a Perceptron with a Boolean output.



Perceptron rule.

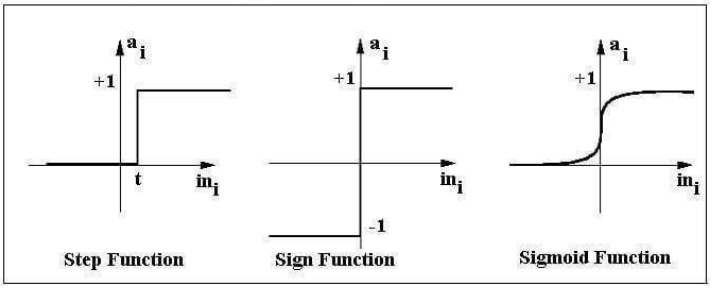
A Boolean output is based on inputs such as salaried, married, age, past credit profile, etc. It has only two values: Yes and No or True and False. The summation function “∑” multiplies all inputs of “x” by weights “w” and then adds them up as follows:

$$w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

In the next section, let us discuss the activation functions of perceptrons.

Activation Functions of Perceptron

The activation function applies a step rule (convert the numerical output into +1 or -1) to check if the output of the weighting function is greater than zero or not.



For example:

If ∑ wixi> 0 => then final output "o" = 1 (issue bank loan)
Else, final output "o" = -1 (deny bank loan)

Step function gets triggered above a certain value of the neuron output; else it outputs zero. Sign Function outputs +1 or -1 depending on whether neuron output is greater than zero or not. Sigmoid is the S-curve and outputs a value between 0 and 1.

Output of Perceptron

Perceptron with a Boolean output:

Inputs: x1…xn
Output: o(x1….xn)

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

Weights: wi=> contribution of input xi to the Perceptron output;
w0=> bias or threshold
If ∑w.x > 0, output is +1, else -1. The neuron gets triggered only when weighted input reaches a certain threshold value.

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

An output of +1 specifies that the neuron is triggered. An output of -1 specifies that the neuron did not get triggered.
"sgn" stands for sign function with output +1 or -1.

6.2.4 Error in Perceptron

In the Perceptron Learning Rule, the predicted output is compared with the known output. If it does not match, the error is propagated backward to allow weight adjustment to happen.

Let us discuss the decision function of Perceptron in the next section.

Perceptron has the following characteristics:

- Perceptron is an algorithm for Supervised Learning of single layer binary linear classifiers.
- Optimal weight coefficients are automatically learned.
- Weights are multiplied with the input features and decision is made if the neuron is fired or not.
- Activation function applies a step rule to check if the output of the weighting function is greater than zero.
- Linear decision boundary is drawn enabling the distinction between the two linearly separable classes +1 and -1.
- If the sum of the input signals exceeds a certain threshold, it outputs a signal; otherwise, there is no output.

Types of activation functions include the sign, step, and sigmoid functions.

Implement Logic Gates with Perceptron

Based on this logic, logic gates can be categorized into seven types:

- AND
- NAND
- OR
- NOR
- NOT
- XOR
- XNOR

Implementing Basic Logic Gates With Perceptron
The logic gates that can be implemented with Perceptron are discussed below.

6.2.4.1 AND

If the two inputs are TRUE (+1), the output of Perceptron is positive, which amounts to TRUE. This is the desired behavior of an AND gate.
x1= 1 (TRUE), x2= 1 (TRUE)
w0 = -.8, w1 = 0.5, w2 = 0.5
=> o(x1, x2) => -.8 + 0.5*1 + 0.5*1 = 0.2 > 0

6.2.4.2 OR

If either of the two inputs are TRUE (+1), the output of Perceptron is positive, which amounts to TRUE.

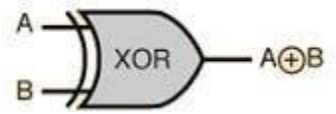
This is the desired behavior of an OR gate.

x1 = 1 (TRUE), x2 = 0 (FALSE)

w0 = -.3, w1 = 0.5, w2 = 0.5
=> o(x1, x2) => -.3 + 0.5*1 + 0.5*0 = 0.2 > 0

6.2.4.3 XOR

A XOR gate, also called as Exclusive OR gate, has two inputs and one output.



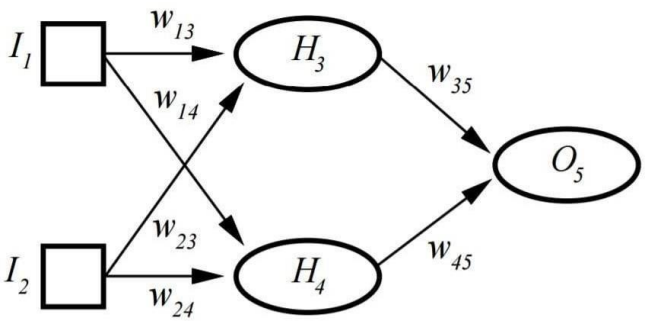
The gate returns a TRUE as the output if and ONLY if one of the input states is true.

XOR Truth Table

Input	Output	
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

XOR Gate with Neural Networks

Unlike the AND and OR gate, an XOR gate requires an intermediate hidden layer for preliminary transformation in order to achieve the logic of an XOR gate.



An XOR gate assigns weights so that XOR conditions are met. It cannot be implemented with a single layer Perceptron and requires Multi-layer Perceptron or MLP.

H represents the hidden layer, which allows XOR implementation.

I1, I2, H3, H4, O5are 0 (FALSE) or 1 (TRUE)

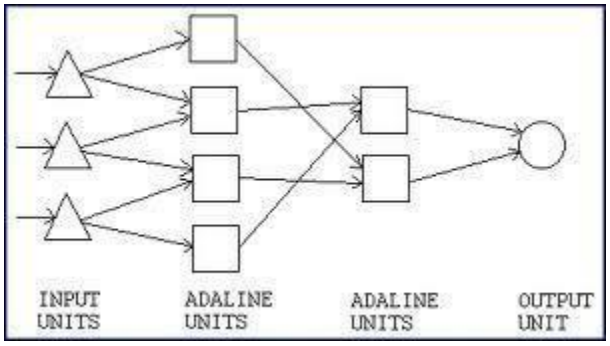
t3= threshold for H3; t4= threshold for H4; t5= threshold for O5

H3= sigmoid (I1*w13+ I2*w23–t3); H4= sigmoid (I1*w14+ I2*w24–t4)

O5= sigmoid (H3*w35+ H4*w45–t5);

6.3 ADALINE

- Known as Adaptive Linear Neuron
- Adaline is a network with a single linear unit
- The Adaline network is trained using the delta rule



Adaline Madaline neural network

6.3.1 Architecture

As already stated Adaline is a single-unit neuron, which receives input from several units and also from one unit, called bias. An Adeline model consists of trainable weights. The inputs are of two values (+1 or -1) and the weights have signs (positive or negative).

Initially random weights are assigned. The net input calculated is applied to a quantizer transfer function (possibly activation function) that restores the output to +1 or -1. The Adaline model compares the actual output with the target output and with the bias and the adjusts all the weights.

6.3.2 Training Algorithm

The Adaline network training algorithm is as follows:

Step0: weights and bias are to be set to some random values but not zero. Set the learning rate parameter α.

Step1: perform steps 2-6 when stopping condition is false.

Step2: perform steps 3-5 for each bipolar training pair s:t

Step3: set activations foe input units i=1 to n.

Step4: calculate the net input to the output unit.

Step5: update the weight and bias for i=1 to n

Step6: if the highest weight change that occurred during training is smaller than a specified tolerance then stop the training process, else continue. This is the test for the stopping condition of a network.

6.3.3 Testing Algorithm

It is very essential to perform the testing of a network that has been trained. When the training has been completed, the Adaline can be used to classify input patterns. A step function is used to test the performance of the network. The testing procedure for the Adaline network is as follows:

Step0: initialize the weights. (The weights are obtained from the training algorithm.)

Step1: perform steps 2-4 for each bipolar input vector x.

Step2: set the activations of the input units to x.

Step3: calculate the net input to the output units

Step4: apply the activation function over the net input calculated.

The following represents the working of Adaline machine learning algorithm based on the above diagram:

- **Net Input function - Combination of Input signals of different strength (weights)**: Input signals of different strength (weights) get combined / added in order to be fed into

the activation function. The combined input or sum of weighted inputs can also be called as **net input.** Pay attention to the Net-input function shown in the above diagram

- **Net input is fed into activation function (Linear)**: Net input is fed into activation function. The activation function of adaline is an identity function. If Z is net input, the identity function would look like \(g(Z) = Z\). The activation function is linear activation function as the output of the function is linear combination of input signals and weights.
- **Activation function output is used to learn weights**: The output of activation function (same as net input owing to identity function) is used to calculate the change in weights related to different inputs which will be updated to learn new weights. Pay attention to feedback loop shown with text **Error or cost.** Recall that in Perceptron, the activation function is a unit step function and the output is binary (1 or 0) based on whether the net input value is greater than or equal to zero (0) or otherwise.
- **Threshold function - Binary prediction (1 or 0) based on unit step function**: The prediction made by Adaline neuron is done in the same manner as in case of Perceptron. The output of activation function, which is net input is compared with 0 and the output is 1 or 0 depending upon whether the net input is greater than or equal to 0. Pay attention in the above diagram as to how the output of activation function is fed into threshold function.

6.3.4 Adaline Python Implementation

The adaline algorithm explained in previous section with the help of diagram will be illustrated further with the help of Python code. Here are the algorithm steps and the related Python implementation:

- **Weighted input signals combined as net input**: The first step is to combine the input signals with respective weights (strength of input signals) and come up with sum of weighted inputs. This is also termed as **net input**.

```
'''

Net Input is sum of weighted input signals

'''

def net_input(self, X):

    weighted_sum = np.dot(X, self.coef_[1:]) + self.coef_[0]

    return weighted_sum
```

- **Activation function invoked with net input**: Net input is fed into activation function to calculate the output. The activation function is a linear activation function. It is an identity function. Thus, \(g(Z) = Z\). Note that the output or return value of activation function is same as input (identity function)

'''

Activation function is fed the net input. As the activation function is

an identity function, the output from activation function is same as the

input to the function.

'''

```
def activation_function(self, X):
    return X
```

- **Prediction based on unit step function**: Prediction is made based on the unit step function which provides binary output as 1 or 0 based on whether the output of the activation function is greater than or equal to zero. If the output of the activation function is greater than or equal to zero, the prediction is 1 or else 0. Note how activation function is invoked with net input and the output of activation function is compared with 0.

'''

Prediction is made based on the output of the activation function

'''

```
def predict(self, X):
  return np.where(self.activation_function(self.net_input(X)) >= 0.0, 1, 0)
```

- **Weights learned using activation function output (continuous value)**: Unlike Perceptron where weights are learned based on the prediction value which is derived as out of unit step function, the weights in case of Adaline is learned by comparing the actual / expected value with the out of activation function which is a **continuous value**. Note that the weights are learned based on **batch gradient descent** algorithm which requires the weights to be updated after considering the weight updates related to all

training examples. This is unlike **stochastic gradient descent** where weights are updated after each training example.

Batch Gradient Descent

1. Weights are updated considering all training examples.
2. Learning of weights can continue for multiple iterations
3. Learning rate needs to be defined

'''

```
def fit(self, X, y):

    rgen = np.random.RandomState(self.random_state)

    self.coef_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])

    for _ in range(self.n_iterations):

        activation_function_output = self.activation_function(self.net_input(X))

        errors = y - activation_function_output

        self.coef_[1:] = self.coef_[1:] + self.learning_rate*X.T.dot(errors)

        self.coef_[0] = self.coef_[0] + self.learning_rate*errors.sum()
```

Here is the entire Python code of Adaline algorithm custom implementation:

```
class CustomAdaline(object):

  def __init__(self, n_iterations=100, random_state=1, learning_rate=0.01):

    self.n_iterations = n_iterations

    self.random_state = random_state

    self.learning_rate = learning_rate

  '''
```

Batch Gradient Descent

1. Weights are updated considering all training examples.
2. Learning of weights can continue for multiple iterations
3. Learning rate needs to be defined

```
'''
def fit(self, X, y):
    rgen = np.random.RandomState(self.random_state)
    self.coef_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
    for _ in range(self.n_iterations):
        activation_function_output = self.activation_function(self.net_input(X))
        errors = y - activation_function_output
        self.coef_[1:] = self.coef_[1:] + self.learning_rate*X.T.dot(errors)
        self.coef_[0] = self.coef_[0] + self.learning_rate*errors.sum()
'''
Net Input is sum of weighted input signals
'''
def net_input(self, X):
        weighted_sum = np.dot(X, self.coef_[1:]) + self.coef_[0]
        return weighted_sum
'''
```

Activation function is fed the net input. As the activation function is an identity function, the output from activation function is same as the

input to the function.

'''

```
def activation_function(self, X):
        return X
```

'''

Prediction is made on the basis of output of activation function

'''

```
def predict(self, X):
    return np.where(self.activation_function(self.net_input(X)) >= 0.0, 1, 0)
```

'''

Model score is calculated based on comparison of

expected value and predicted value

'''

```
def score(self, X, y):
    misclassified_data_count = 0
    for xi, target in zip(X, y):
        output = self.predict(xi)
        if(target != output):
            misclassified_data_count += 1
    total_data_count = len(X)
```

```
self.score_ = (total_data_count - misclassified_data_count)/total_data_count

return self.score_
```

6.4 CONCLUSIONS

Here is the summary of what you learned in this post about Adaline algorithm and its Python implementation:

- Adaline algorithm mimics a neuron in the human brain
- Adaline is similar to the algorithm Perceptron. It can also be termed as a single-layer neural network.
- The difference between Adaline and Perceptron lies in the manner in which weights are learned based on the differences between the output label and the continuous value output of the activation function. In Perceptron, the difference between an actual label and a predicted label is used to learn the weights.

UNIT VII: MULTIPLE ADAPTIVE LINEAR NEURON (MADALINE)

STRUCTURE

7.0 Objectives

7.1 Introduction

- **7.1.1 Architecture**
- **7.1.2 Training Algorithm**

7.2 Backpropagation Algorithm

- **7.2.1 Wheat Seeds Dataset**

7.3 Initialize Network

- **7.3.1 Forward Propagate**
- **7.3.2 Neuron Activation**
- **7.3.3 Neuron Transfer**
- **7.3.4 Forward Propagation**
- **7.3.5 Back Propagate Error**

7.4 Train Network

7.5 Predict

7.6 Backpropagation Algorithm to the Wheat Seeds Dataset

7.7 Summary

7.7 Practice Excercises

7.0 OBJECTIVES

- Understanding MADALINE, ITS : architecture, algorithm, application for XOR problem
- Understanding Backpropagation Neural Network, its architecture, parameters, algorithm, applications and different issues regarding convergence

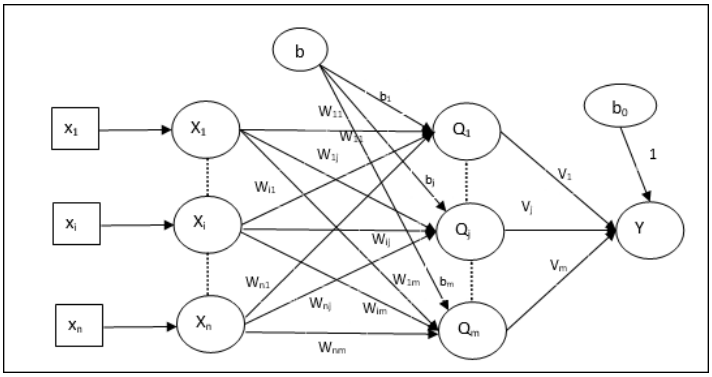
7.1 INTRODUCTION

Madaline which stands for Multiple Adaptive Linear Neuron is a network that consists of many Adalines in parallel. It will have a single output unit. Some important points about Madaline are as follows −

- It is just like a multilayer perceptron, where Adaline will act as a hidden unit between the input and the Madaline layer.
- The weights and the bias between the input and Adaline layers, as we see in the Adaline architecture, are adjustable.
- The Adaline and Madaline layers have fixed weights and a bias of 1.
- Training can be done with the help of the Delta rule.

7.1.1 Architecture

The architecture of Madaline consists of **"n"** neurons of the input layer, **"m"** neurons of the Adaline layer, and 1 neuron of the Madaline layer. The Adaline layer can be considered as the hidden layer as it is between the input layer and the output layer, i.e. the Madaline layer.



7.1.2 Training Algorithm

By now we know that only the weights and bias between the input and the Adaline layer are to be adjusted, and the weights and bias between the Adaline and the Madaline layer are fixed.

Step 1 − Initialize the following to start the training −

- Weights

- Bias
- Learning rate αα

For easy calculation and simplicity, weights and bias must be set equal to 0 and the learning rate must be set equal to 1.

Step 2 − Continue steps 3-8 when the stopping condition is not true.

Step 3 − Continue step 4-7 for every bipolar training pair **s:t**.

Step 4 − Activate each input unit as follows −

xi=si(i=1ton)xi=si(i=1ton)

Step 5 − Obtain the net input at each hidden layer, i.e. the Adaline layer with the following relation −

Qinj=bj+∑inxiwijj=1tomQinj=bj+∑inxiwijj=1tom

Here **„b"** is bias and **„n"** is the total number of input neurons.

Step 6 − Apply the following activation function to obtain the final output at the Adaline and the Madaline layer −

f(x)={1−1ifx⩾0ifx<0f(x)={1ifx⩾0−1ifx<0

Output at the hidden (Adaline) unit

Qj=f(Qinj)Qj=f(Qinj)

Final output of the network

y=f(yin)y=f(yin)

i.e. yinj=b0+∑mj=1Qjvjyinj=b0+∑j=1mQjvj

Step 7 − Calculate the error and adjust the weights as follows −

Case 1 − if **y ≠ t** and **t = 1** then,

wij(new)=wij(old)+α(1−Qinj)xiwij(new)=wij(old)+α(1−Qinj)xi

bj(new)=bj(old)+α(1−Qinj)bj(new)=bj(old)+α(1−Qinj)

In this case, the weights would be updated on $\mathbf{Q_j}$ where the net input is close to 0 because **t = 1**.

Case 2 − if **y ≠ t** and **t = -1** then,

wik(new)=wik(old)+α(−1−Qink)xiwik(new)=wik(old)+α(−1−Qink)xi

bk(new)=bk(old)+α(−1−Qink)bk(new)=bk(old)+α(−1−Qink)

In this case, the weights would be updated on $\mathbf{Q_k}$ where the net input is positive because **t = -1**.

Here **„y"** is the actual output and **„t"** is the desired/target output.

Case 3 − if **y = t** then

There would be no change in weights.

Step 8 − Test for the stopping condition, which will happen when there is no change in weight or the highest weight change occurred during training is smaller than the specified tolerance.

7.2 BACKPROPAGATION ALGORITHM

The Backpropagation algorithm is a supervised learning method for multilayer feedforward networks from the field of Artificial Neural Networks. Feed-forward neural networks are inspired by the information processing of one or more neural cells, called a neuron. A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. The axon carries the signal out to synapses, which are the connections of a cell's axon to other cell's dendrites.

The principle of the backpropagation approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal. The system is trained using a supervised learning method, where the error between the system's output and a known expected output is presented to the system and used to modify its internal state.

Technically, the backpropagation algorithm is a method for training the weights in a multilayer feed-forward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. A standard network structure is one input layer, one hidden layer, and one output layer.

Backpropagation can be used for both classification and regression problems, but we will focus on classification in this tutorial. In classification problems, best results are achieved when the network has one neuron in the output layer for each class value. For example, a 2-class or binary classification problem with the class values of A and B. These expected outputs would have to be transformed into binary vectors with one column for each class value. Such as [1, 0] and [0, 1] for A and B respectively. This is called a one hot encoding.

7.2.1 Wheat Seeds Dataset

The seeds dataset involves the prediction of species given measurements seeds from different varieties of wheat.

There are 201 records and 7 numerical input variables. It is a classification problem with 3 output classes. The scale for each numeric input value vary, so some data normalization may be required for use with algorithms that weight inputs like the backpropagation algorithm.

Below is a sample of the first 5 rows of the dataset.

1 15.26,14.84,0.871,5.763,3.312,2.221,5.22,1

2 14.88,14.57,0.8811,5.554,3.333,1.018,4.956,1

3 14.29,14.09,0.905,5.291,3.337,2.699,4.825,1

4 13.84,13.94,0.8955,5.324,3.379,2.259,4.805,1

5 16.14,14.99,0.9034,5.658,3.562,1.355,5.175,1

Using the Zero Rule algorithm that predicts the most common class value, the baseline accuracy for the problem is 28.095%.

You can learn more and download the seeds dataset from the UCI Machine Learning Repository. Download the seeds dataset and place it into your current working directory with the filename **seeds_dataset.csv**.This section is broken down into 6 parts:

1. Initialize Network.
2. Forward Propagate.
3. Back Propagate Error.
4. Train Network.
5. Predict.
6. Seeds Dataset Case Study.

These steps will provide the foundation that you need to implement the backpropagation algorithm from scratch and apply it to your own predictive modeling problems.

7.3 INITIALIZE NETWORK

Let's start with something easy, the creation of a new network ready for training.

Each neuron has a set of weights that need to be maintained. One weight for each input connection and an additional weight for the bias. We will need to store additional properties for a neuron during training, therefore we will use a dictionary to represent each neuron and store properties by names such as '**weights**' for the weights.
A network is organized into layers. The input layer is really just a row from our training dataset. The first real layer is the hidden layer. This is followed by the output layer that has one neuron for each class value.

We will organize layers as arrays of dictionaries and treat the whole network as an array of layers.

It is good practice to initialize the network weights to small random numbers. In this case, will we use random numbers in the range of 0 to 1.

Below is a function named **initialize_network()** that creates a new neural network ready for training. It accepts three parameters, the number of inputs, the number of neurons to have in the hidden layer and the number of outputs.
You can see that for the hidden layer we create **n_hidden** neurons and each neuron in the hidden layer has **n_inputs + 1** weights, one for each input column in a dataset and an additional one for the bias.
You can also see that the output layer that connects to the hidden layer has **n_outputs** neurons, each with **n_hidden + 1** weights. This means that each neuron in the output layer connects to (has a weight for) each neuron in the hidden layer.

```
# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
	network = list()
	hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
	network.append(hidden_layer)
	output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
	network.append(output_layer)
	return network
```

Let's test out this function. Below is a complete example that creates a small network.

```
from random import seed
from random import random

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
	network = list()
	hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
	network.append(hidden_layer)
	output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in
```

```
range(n_outputs)]
        network.append(output_layer)
        return network


seed(1)
network = initialize_network(2, 1, 2)
for layer in network:
        print(layer)
```

Running the example, you can see that the code prints out each layer one by one. You can see the hidden layer has one neuron with 2 input weights plus the bias. The output layer has 2 neurons, each with 1 weight plus the bias.

```
1 [{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}]
2 [{'weights': [0.2550690257394217, 0.4954350870919409 5]}, {'weights':
  [0.4494910647887381, 0.651592972722763]}]
```

Now that we know how to create and initialize a network, let's see how we can use it to calculate an output.

7.3.1 Forward Propagate

We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values.

We call this forward-propagation.

It is the technique we will need to generate predictions during training that will need to be corrected, and it is the method we will need after the network is trained to make predictions on new data.

We can break forward propagation down into three parts:

1. Neuron Activation.
2. Neuron Transfer.
3. Forward Propagation.

7.3.2 Neuron Activation

The first step is to calculate the activation of one neuron given an input.

The input could be a row from our training dataset, as in the case of the hidden layer. It may also be the outputs from each neuron in the hidden layer, in the case of the output layer.

Neuron activation is calculated as the weighted sum of the inputs. Much like linear regression.

1 activation = sum(weight_i * input_i) + bias

Where **weight** is a network weight, **input** is an input, **i** is the index of a weight or an input and **bias** is a special weight that has no input to multiply with (or you can think of the input as always being 1.0).
Below is an implementation of this in a function named **activate()**. You can see that the function assumes that the bias is the last weight in the list of weights. This helps here and later to make the code easier to read.

```
# Calculate neuron activation for an input

def activate(weights, inputs):

    activation = weights[-1]

    for i in range(len(weights)-1):

        activation += weights[i] * inputs[i]

    return activation
```

Now, let's see how to use the neuron activation.

7.3.3 Neuron Transfer

Once a neuron is activated, we need to transfer the activation to see what the neuron output actually is.

Different transfer functions can be used. It is traditional to use the sigmoid activation function, but you can also use the tanh (hyperbolic tangent) function to transfer outputs. More recently, the rectifier transfer function has been popular with large deep learning networks.
The sigmoid activation function looks like an S shape, it's also called the logistic function. It can take any input value and produce a number between 0 and 1 on an S-curve. It is also a function of which we can easily calculate the derivative (slope) that we will need later when backpropagating error.

We can transfer an activation function using the sigmoid function as follows:

output = 1 / (1 + e^(-activation))

Where **e** is the base of the natural logarithms (Euler's number).
Below is a function named **transfer()** that implements the sigmoid equation.

```
# Transfer neuron activation

def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))
```

Now that we have the pieces, let's see how they are used.

7.3.4 Forward Propagation

Forward propagating an input is straightforward.
We work through each layer of our network calculating the outputs for each neuron. All of the outputs from one layer become inputs to the neurons on the next layer.

Below is a function named **forward_propagate()** that implements the forward propagation for a row of data from our dataset with our neural network.
You can see that a neuron's output value is stored in the neuron with the name '**output**'. You can also see that we collect the outputs for a layer in an array named **new_inputs** that becomes the array **inputs** and is used as inputs for the following layer.
The function returns the outputs from the last layer also called the output layer.

```
# Forward propagate input to a network output

def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs
```

Let's put all of these pieces together and test out the forward propagation of our network.

Running the example propagates the input pattern [1, 0] and produces an output value that is printed. Because the output layer has two neurons, we get a list of two numbers as output.

The actual output values are just nonsense for now, but next, we will start to learn how to make the weights in the neurons more useful.

[0.6629970129852887, 0.7253160725279748]

7.3.5 Back Propagate Error

The backpropagation algorithm is named for how weights are trained.

Error is calculated between the expected outputs and the outputs forward propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go.

The math for backpropagating error is rooted in calculus, but we will remain high level in this section and focus on what is calculated and how rather than why the calculations take this particular form.

This part is broken down into two sections.

1. Transfer Derivative.
2. Error Backpropagation.

7.3.5.1 Transfer Derivative

Given an output value from a neuron, we need to calculate its slope.

We are using the sigmoid transfer function, the derivative of which can be calculated as follows:

derivative = output * (1.0 - output)

Below is a function named **transfer_derivative()** that implements this equation.

```
# Calculate the derivative of an neuron output
def transfer_derivative(output):
        return output * (1.0 - output)
```

Now, let’s see how this can be used.

7.3.5.2 Error Backpropagation

The first step is to calculate the error for each output neuron, this will give us our error signal (input) to propagate backwards through the network.

The error for a given neuron can be calculated as follows:

error = (expected - output) * transfer_derivative(output)

Where **expected** is the expected output value for the neuron, **output** is the output value for the neuron and **transfer_derivative()** calculates the slope of the neuron's output value, as shown above.
This error calculation is used for neurons in the output layer. The expected value is the class value itself. In the hidden layer, things are a little more complicated.

The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in the output layer. Think of the error traveling back along the weights of the output layer to the neurons in the hidden layer.

The back-propagated error signal is accumulated and then used to determine the error for the neuron in the hidden layer, as follows:

error = (weight_k * error_j) * transfer_derivative(output)

Where **error_j** is the error signal from the **j**th neuron in the output layer, **weight_k** is the weight that connects the **k**th neuron to the current neuron and output is the output for the current neuron.
Below is a function named **backward_propagate_error()** that implements this procedure.
You can see that the error signal calculated for each neuron is stored with the name 'delta'. You can see that the layers of the network are iterated in reverse order, starting at the output and working backward. This ensures that the neurons in the output layer have 'delta' values calculated first that neurons in the hidden layer can use in the subsequent iteration. I chose the name 'delta' to reflect the change the error implies on the neuron (e.g. the weight delta).

You can see that the error signal for neurons in the hidden layer is accumulated from neurons in the output layer where the hidden neuron number **j** is also the index of the neuron's weight in the output layer **neuron[„weights"][j]**.

```
# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network)-1:
            for j in range(len(layer)):
```

```
                    error = 0.0
                    for neuron in network[i + 1]:
                        error += (neuron['weights'][j] * neuron['delta'])
                    errors.append(error)
            else:
                for j in range(len(layer)):
                    neuron = layer[j]
                    errors.append(expected[j] - neuron['output'])
            for j in range(len(layer)):
                neuron = layer[j]
                neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
```

Let's put all of the pieces together and see how it works.

Explore yourself by combining all together.

We define a fixed neural network with output values and backpropagate an expected output pattern. The complete example is listed below.

```
# Calculate the derivative of a neuron output
def transfer_derivative(output):
    return output * (1.0 - output)


# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network)-1:
            for j in range(len(layer)):
```

```
                    error = 0.0
                    for neuron in network[i + 1]:
                        error += (neuron['weights'][j] * neuron['delta'])
                    errors.append(error)
            else:
                for j in range(len(layer)):
                    neuron = layer[j]
                    errors.append(expected[j] - neuron['output'])
            for j in range(len(layer)):
                neuron = layer[j]
                neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])


# test backpropagation of error
network = [[{'output': 0.7105668883115941, 'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}],
                [{'output': 0.6213859615555266, 'weights': [0.2550690257394217, 0.49543508709194095]}, {'output': 0.6573693455986976, 'weights': [0.4494910647887381, 0.651592972722763]}]]
expected = [0, 1]
backward_propagate_error(network, expected)
for layer in network:
    print(layer)
```

Running the example prints the network after the backpropagation of error is complete. You can see that error values are calculated and stored in the neurons for the output layer and the hidden layer.

```
1 [{'output': 0.7105668883115941, 'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614], 'delta': -0.0005348048046610517}]
2 [{'output': 0.6213859615555266, 'weights': [0.2550690257394217, 0.49543508709194095], 'delta': -0.14619064683582808}, {'output': 0.6573693455986976, 'weights':
```

[0.4494910647887381, 0.651592972722763], 'delta': 0.0771723774346327}]

Now let's use the backpropagation of error to train the network.

7.4 TRAIN NETWORK

The network is trained using stochastic gradient descent.

This involves multiple iterations of exposing a training dataset to the network and for each row of data forward propagating the inputs, backpropagating the error and updating the network weights.

This part is broken down into two sections:

1. Update Weights.
2. Train Network.

7.4.1 Update Weights

Once errors are calculated for each neuron in the network via the back propagation method above, they can be used to update weights.

Network weights are updated as follows:

weight = weight + learning_rate * error * input

Where **weight** is a given weight, **learning_rate** is a parameter that you must specify, **error** is the error calculated by the backpropagation procedure for the neuron and **input** is the input value that caused the error.
The same procedure can be used for updating the bias weight, except there is no input term, or input is the fixed value of 1.0.

Learning rate controls how much to change the weight to correct for the error. For example, a value of 0.1 will update the weight 10% of the amount that it possibly could be updated. Small learning rates are preferred that cause slower learning over a large number of training iterations. This increases the likelihood of the network finding a good set of weights across all layers rather than the fastest set of weights that minimize error (called premature convergence).

Below is a function named **update_weights()** that updates the weights for a network given an input row of data, a learning rate and assume that a forward and backward propagation have already been performed.
Remember that the input for the output layer is a collection of outputs from the hidden layer.

Update network weights with error

```
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] += l_rate * neuron['delta']
```

Now we know how to update network weights, let's see how we can do it repeatedly.

7.4.2 Train Network

As mentioned, the network is updated using stochastic gradient descent.

This involves first looping for a fixed number of epochs and within each epoch updating the network for each row in the training dataset.

Because updates are made for each training pattern, this type of learning is called online learning. If errors were accumulated across an epoch before updating the weights, this is called batch learning or batch gradient descent.

Below is a function that implements the training of an already initialized neural network with a given training dataset, learning rate, fixed number of epochs and an expected number of output values.

The expected number of output values is used to transform class values in the training data into a one hot encoding. That is a binary vector with one column for each class value to match the output of the network. This is required to calculate the error for the output layer.

You can also see that the sum squared error between the expected output and the network output is accumulated each epoch and printed. This is helpful to create a trace of how much the network is learning and improving each epoch.

```
# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
```

```
    for epoch in range(n_epoch):
        sum_error = 0
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            sum_error += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
```

We now have all of the pieces to train the network. We can put together an example that includes everything we've seen so far including network initialization and train a network on a small dataset.

Below is a small contrived dataset that we can use to test out training our neural network.

X1	X2	Y
2.7810836	2.550537003	0
1.465489372	2.362125076	0
3.396561688	4.400293529	0
1.38807019	1.850220317	0
3.06407232	3.005305973	0
7.627531214	2.759262235	1
5.332441248	2.088626775	1
6.922596716	1.77106367	1
8.675418651	-0.242068655	1
7.673756466	3.508563011	1

Below is the complete example. We will use 2 neurons in the hidden layer. It is a binary classification problem (2 classes) so there will be two neurons in the output layer. The network

will be trained for 20 epochs with a learning rate of 0.5, which is high because we are training for so few iterations.

Running the example first prints the sum squared error each training epoch. We can see a trend of this error decreasing with each epoch.

Once trained, the network is printed, showing the learned weights. Also still in the network are output and delta values that can be ignored. We could update our training function to delete these data if we wanted.

```
1  >epoch=0, lrate=0.500, error=6.350
2  >epoch=1, lrate=0.500, error=5.531
3  >epoch=2, lrate=0.500, error=5.221
4  >epoch=3, lrate=0.500, error=4.951
5  >epoch=4, lrate=0.500, error=4.519
6  >epoch=5, lrate=0.500, error=4.173
7  >epoch=6, lrate=0.500, error=3.835
8  >epoch=7, lrate=0.500, error=3.506
9  >epoch=8, lrate=0.500, error=3.192
10 >epoch=9, lrate=0.500, error=2.898
11 >epoch=10, lrate=0.500, error=2.626
12 >epoch=11, lrate=0.500, error=2.377
13 >epoch=12, lrate=0.500, error=2.153
14 >epoch=13, lrate=0.500, error=1.953
15 >epoch=14, lrate=0.500, error=1.774
16 >epoch=15, lrate=0.500, error=1.614
17 >epoch=16, lrate=0.500, error=1.472
18 >epoch=17, lrate=0.500, error=1.346
19 >epoch=18, lrate=0.500, error=1.233
20 >epoch=19, lrate=0.500, error=1.132
21 [{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output':
```

22 0.029980305604426185, 'delta': -0.0059546604162323625}, {'weights': [0.37711098142462157, -0.0625909894552989, 0.2765123702642716], 'output': 0.9456229000211323, 'delta': 0.0026279652850863837}]

[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output': 0.23648794202357587, 'delta': -0.04270059278364587}, {'weights': [-2.5584149848484263, 1.0036422106209202, 0.42383086467582715], 'output': 0.7790535202438367, 'delta': 0.03803132596437354}]

Once a network is trained, we need to use it to make predictions.

7.5 PREDICT

Making predictions with a trained neural network is easy enough.

We have already seen how to forward-propagate an input pattern to get an output. This is all we need to do to make a prediction. We can use the output values themselves directly as the probability of a pattern belonging to each output class.

It may be more useful to turn this output back into a crisp class prediction. We can do this by selecting the class value with the larger probability. This is also called the arg max function.
Below is a function named **predict()** that implements this procedure. It returns the index in the network output that has the largest probability. It assumes that class values have been converted to integers starting at 0.

```
# Make a prediction with a network

def predict(network, row):

    outputs = forward_propagate(network, row)

    return outputs.index(max(outputs))
```

We can put this together with our code above for forward propagating input and with our small contrived dataset to test making predictions with an already-trained network. The example hardcodes a network trained from the previous step.

It shows that the network achieves 100% accuracy on this small dataset.

7.6 BACKPROPAGATION ALGORITHM TO THE WHEAT SEEDS DATASET

The first step is to load the dataset and convert the loaded data to numbers that we can use in our neural network. For this we will use the helper function **load_csv()** to load the file, **str_column_to_float()** to convert string numbers to floats and **str_column_to_int()** to convert the class column to integer values.

Input values vary in scale and need to be normalized to the range of 0 and 1. It is generally good practice to normalize input values to the range of the chosen transfer function, in this case, the sigmoid function that outputs values between 0 and 1.
The **dataset_minmax()** and **normalize_dataset()** helper functions were used to normalize the input values.
We will evaluate the algorithm using k-fold cross-validation with 5 folds. This means that 201/5=40.2 or 40 records will be in each fold. We will use the helper functions **evaluate_algorithm()** to evaluate the algorithm with cross-validation and **accuracy_metric()** to calculate the accuracy of predictions.
A new function named **back_propagation()** was developed to manage the application of the Backpropagation algorithm, first initializing a network, training it on the training dataset and then using the trained network to make predictions on a test dataset.
The complete example is listed below.

```
# Backprop on the Seeds Dataset
from random import seed
from random import randrange
from random import random
from csv import reader
from math import exp

# Load a CSV file
def load_csv(filename):
	dataset = list()
	with open(filename, 'r') as file:
		csv_reader = reader(file)
		for row in csv_reader:
			if not row:
				continue
			dataset.append(row)
	return dataset
```

```
# Convert string column to float
def str_column_to_float(dataset, column):
	for row in dataset:
		row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
	class_values = [row[column] for row in dataset]
	unique = set(class_values)
	lookup = dict()
	for i, value in enumerate(unique):
		lookup[value] = i
	for row in dataset:
		row[column] = lookup[row[column]]
	return lookup

# Find the min and max values for each column
def dataset_minmax(dataset):
	minmax = list()
	stats = [[min(column), max(column)] for column in zip(*dataset)]
	return stats

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
	for row in dataset:
```

```
            for i in range(len(row)-1):
                row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for i in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
```

```
	folds = cross_validation_split(dataset, n_folds)
	scores = list()
	for fold in folds:
		train_set = list(folds)
		train_set.remove(fold)
		train_set = sum(train_set, [])
		test_set = list()
		for row in fold:
			row_copy = list(row)
			test_set.append(row_copy)
			row_copy[-1] = None
		predicted = algorithm(train_set, test_set, *args)
		actual = [row[-1] for row in fold]
		accuracy = accuracy_metric(actual, predicted)
		scores.append(accuracy)
	return scores

# Calculate neuron activation for an input
def activate(weights, inputs):
	activation = weights[-1]
	for i in range(len(weights)-1):
		activation += weights[i] * inputs[i]
	return activation

# Transfer neuron activation
def transfer(activation):
```

```
        return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
        inputs = row
        for layer in network:
                new_inputs = []
                for neuron in layer:
                        activation = activate(neuron['weights'], inputs)
                        neuron['output'] = transfer(activation)
                        new_inputs.append(neuron['output'])
                inputs = new_inputs
        return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
        return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
        for i in reversed(range(len(network))):
                layer = network[i]
                errors = list()
                if i != len(network)-1:
                        for j in range(len(layer)):
                                error = 0.0
```

```
                    for neuron in network[i + 1]:
                        error += (neuron['weights'][j] * neuron['delta'])
                    errors.append(error)
            else:
                for j in range(len(layer)):
                    neuron = layer[j]
                    errors.append(expected[j] - neuron['output'])
            for j in range(len(layer)):
                neuron = layer[j]
                neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])

# Update network weights with error
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] += l_rate * neuron['delta']

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
    for epoch in range(n_epoch):
        for row in train:
```

```
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    return network

# Make a prediction with a network
def predict(network, row):
    outputs = forward_propagate(network, row)
    return outputs.index(max(outputs))

# Backpropagation Algorithm With Stochastic Gradient Descent
def back_propagation(train, test, l_rate, n_epoch, n_hidden):
    n_inputs = len(train[0]) - 1
    n_outputs = len(set([row[-1] for row in train]))
    network = initialize_network(n_inputs, n_hidden, n_outputs)
```

```
        train_network(network, train, l_rate, n_epoch, n_outputs)
        predictions = list()
        for row in test:
                prediction = predict(network, row)
                predictions.append(prediction)
        return(predictions)

# Test Backprop on Seeds dataset
seed(1)
# load and prepare data
filename = 'seeds_dataset.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
        str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# normalize input variables
minmax = dataset_minmax(dataset)
normalize_dataset(dataset, minmax)
# evaluate algorithm
n_folds = 5
l_rate = 0.3
n_epoch = 500
n_hidden = 5
scores = evaluate_algorithm(dataset, back_propagation, n_folds, l_rate, n_epoch, n_hidden)
print('Scores: %s' % scores)
```

```
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

A network with 5 neurons in the hidden layer and 3 neurons in the output layer was constructed. The network was trained for 500 epochs with a learning rate of 0.3. These parameters were found with a little trial and error, but you may be able to do much better.

Running the example prints the average classification accuracy on each fold as well as the average performance across all folds.

You can see that backpropagation and the chosen configuration achieved a mean classification accuracy of about 93% which is dramatically better than the Zero Rule algorithm that did slightly better than 28% accuracy.

```
Scores: [92.85714285714286, 92.85714285714286, 97.61904761904762,
92.857142857142816, 90.47619047619048]

Mean Accuracy: 93.333%
```

MACHINE LEARNING
UNIT VIII: UNSUPERVISED MODELS

STRUCTURE

8.0 Objectives

8.1 Kohonen Self – Organizing Maps and its Architecture

8.2 Learning process of Self-organizing Maps (SOM)

8.3 Implementation of Self-organizing Maps

- **8.3.1 Implementation with Python and Tensorflow**
- **8.3.2 Initialization**
- **8.3.3 BMU Calculations**
- **8.3.4 Update Weights**
- **8.3.5 Usage**

8.4 ART Introduction

- **8.4.1 Operating Principal**

8.5 ART 1

8.6 Algorithm

8.7 ART1 Implementation Process

8.8 Summary

8.9 Practice Questions

8.0 Objectives

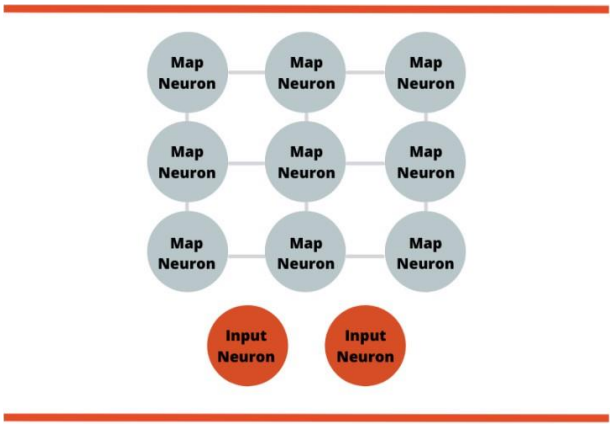
Understanding unsupervised models: Kohonen Self –Organizing Maps, its architecture, algorithm, application. Adaptive Resonance Theory, Its basic basic architecture and operation

8.1 KOHONEN SELF –ORGANIZING MAPS AND ITS ARCHITECTURE

Despite the fact that the early ideas for this sort of network can be followed back to 1981, they were created and formalized in 1992 by Teuvo Kohonen, an educator of the Academy of Finland. In a quintessence, they are utilizing vector quantization to distinguish designs in multidimensional information and address them in many lower-dimensional spaces.

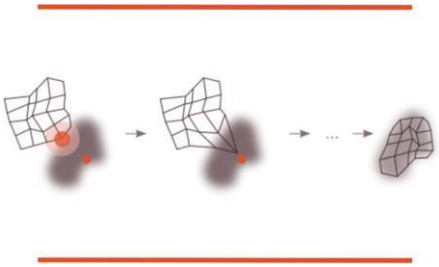
However it may take a new viewpoint on these networks and overlook standard neuron/association and loads ideas. These networks are utilizing similar terms however they have an alternate significance in their domain.

These networks are also known as maps. They resembles as sheet-like neural network, in which neurons are initiated by different patterns in input signals. Investigate the image beneath:



Here we can see a straightforward self-organizing structure. The features are represented by two input neurons. This additionally implies that our input data can be addressed by three-dimensional vectors. Above them, map neurons are there. The objective of these neurons is to introduce information expected on input neurons as two-dimensional information. This means, that in this model self-organizing map utilizes unsupervised learning to convert 3-D data to 2-D representation

We can have quite a few dimensions in our provided data and quite a few dimensions for our yield (expected) information. Notice that each map neuron is associated with each input neuron and that map neurons are not associated with one another. SO the adjacent map neurons are unaware of their neighbor values



Every association has a weight connected to it, yet they are not utilized similarly as in feedforward networks. Fundamentally, you can see that weights are presently addressing the association of the mapping neuron with the input vector. Each map neuron can be distinguished by remarkable I, j coordinates, and weights on their associations are refreshed dependent on the qualities on the information, yet inclining further toward that later.

Presently you can perceive any reason why take a new point of view on this kind of network. Although they are utilizing similar terms, similar to neurons, associations, and weights their significance is unique. Aside from that, you can see that their structure is less difficult than the construction of the other feed-forward neural networks resulting in different learning processes. In the following section learning of networks are there.

8.2 LEARNING PROCESS OF SELF-ORGANIZING MAPS (SOM)

As we referenced already, self-organizing maps utilize unsupervised learning. This sort of learning is also known as competitive learning. The initial phase in the learning system of self-organizing maps is the configuration of the weight on associations.

From that point forward, an arbitrary example from the dataset is utilized as a contribution to the network. The network then, at that point, ascertains weights of which neurons are most similar to the information (input vector).

$$Distance^2 = \sum_{i=0}^{n} (input_i - weight_i)^2$$

where *n* is denoted as weights. The map neuron with the outcome is called Best Matching Unit or BMU. This implies that the input vector can be addressed with this mapping neuron. Presently, self-organizing maps are not simply computing this point during the learning system, however, they additionally attempt to make it "closer" to the got input information.

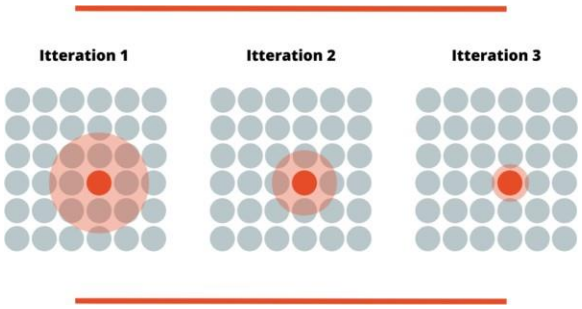
This implies that weights on this association are refreshed in a way that the determined distance is much more modest. In any case, that isn't the main thing that is finished. The neighbor's weight of BMU is additionally adjusted so they are nearer to this input vector as well. This is the way the entire map is 'pulled' toward this point. For this reason, we need to know the span of the neighbors that will be refreshed. This range is at first enormous, yet it is decreased in each cycle (epoch). In this way, the subsequent stage in preparing self-organizing maps is working out referenced radius values. The accompanying equation is applied:

$$\sigma(t) = \sigma_0 e^{-\frac{t}{\lambda}}$$

where *t* is the current cycle, *σo* is the map radius. The λ is defined as:

$$\lambda = k/\sigma_0$$

where *k* is the number of repetitions. This formula uses exponential decay, making the span more modest as the training goes on, which was the underlying objective. This implies that each cycle through the information will carry significant points nearer to the input data in this way the self-organizing maps are fine-tuned.



The updation of weights is there when the radius of the current cycle is determined. The nearer the neuron is to the BMU the possibility of changing the weight's increases. This is accomplished by utilizing this equation:

$$weight(t+1) = weight(t) + \Theta(t)L(t)(input(t) - weight(t))$$

This is the fundamental learning formula, and it has a couple of significant focuses that ought to be discussed. The first is L(t) which addresses the learning rate. Additionally for radius formula, it is using exponential decay and it is getting more modest in each cycle:

$$L(t) = L_0 e^{-\frac{t}{\lambda}}$$

Aside from that, as the weights are directly the neuron closer to BMU, In the equation, that is taken care of with the Θ(t). This value is determined this way:

$$\Theta(t) = e^{-distBMU/2\sigma(t)^2}$$

If the neuron is nearer to the BMU, distBMU is more modest, and with that Θ(t) esteem is more like 1. This implies that the values of the neuron weight will be more changed. This entire method is repeated a few times.

To summarize it, these are simply the main steps in the self-organizing map learning process:

1. Initialization of weights
2. The input for the network is selected from the input vector in the dataset
3. BMU is determined
4. The radius of neighbors that will be refreshed is determined
5. Each neuron's weight inside the span are adapted to make them more similar to the input vector
6. Steps from 2 to 5 are repeated for each information vector of the dataset

There are a ton of varieties of the situations introduced utilized in the learning system of self-organizing maps. Indeed, a great deal of research has been finished attempting to get to the ideal value for the number of iteration, the learning rate, and the local span. The innovator, Teuvo Kohonen, recommended that this learning system ought to be parted into two stages. During the initial stage, the learning rate would be decreased from 0.9 to 0.1 and the local range from a large portion of the width of the lattice to the promptly surrounding nodes.

In the subsequent stage, the learning rate would be additionally diminished from 0.1 to 0.0. Notwithstanding, there would be twofold or more repetition in the subsequent stage and the neighborhood value ought to stay fixed at 1, which means the BMU as it were. This implies that the initial stage would be utilized for learning and the subsequent stage would be utilized for fine-tuning.

8.3 IMPLEMENTATIONS OF SELF-ORGANIZING MAPS (SOM)

Mini-SOM is a minimalistic, *Numpy* based execution of the Self-Organizing Maps and it is very easy to understand. It can be installed using *pip:*

pip install minisom

or using the downloaded setup:

python setup.py install

As referenced, the use of this library is very simple and clear. You have to create a SOM class object and defining its various attributes such as learning rate, size, radius, and size of the input. From that point forward, you can utilize one of the two alternatives for training that this execution gives – train_batch or train_random. The first uses ordered samples, while the subsequent one mixes through the samples. Here is an example:

```
from minisom import MiniSom
som = MiniSom(4, 4, 2, sigma=0.5, learning_rate=0.5)
som.train_random(data, 50)
```

In this model, 4×4 Self-Organizing Map is made, with the 2 input nodes. Learning rate and range (sigma) are both introduced to 0.5. Then, Self-Organizing Map is prepared with input information for 50 repetitions utilizing train_random.

8.3.1 Implementation with Python and Tensorflow

For this execution, a low-level API of TensorFlow is utilized. Here you can track down a fast aide on the most proficient method to rapidly introduce it and how to begin functioning with it. As a rule, the low-level API of this library is utilized for the execution. Thus, we should look at the code:

```
import tensorflow as tf
import numpy as np


class SOM(object):
    def __init__(self, x, y, input_dim, learning_rate, radius, num_iter=111):

        #Initialize properties
        self._x = x
        self._y = y
        self._learning_rate = float(learning_rate)
```

```
        self._radius = float(radius)
        self._num_iter = num_iter
        self._graph = tf.Graph()

        #Initialize graph
        with self._graph.as_default():

            #Initializing variables and placeholders
            self._weights = tf.Variable(tf.random_normal([x*y, input_dim]))
            self._locations = self._generate_index_matrix(x, y)
            self._input = tf.placeholder("float", [input_dim])
            self._iter_input = tf.placeholder("float")

            #Calculating BMU
            input_matix = tf.stack([self._input for i in range(x*y)])
            distances = tf.sqrt(tf.reduce_sum(tf.pow(tf.subtract(self._weights, input_matix), 2), 1))
            bmu = tf.argmin(distances, 0)

            #Get BMU location
            mask = tf.pad(tf.reshape(bmu, [1]), np.array([[0, 1]]))
            size = tf.cast(tf.constant(np.array([1, 2])), dtype=tf.int64)
            bmu_location = tf.reshape(tf.slice(self._locations, mask, size), [2])

            #Calculate learning rate and radius
            decay_function = tf.subtract(1.0, tf.div(self._iter_input, self._num_iter))
            _current_learning_rate = tf.multiply(self._learning_rate, decay_function)
            _current_radius = tf.multiply(self._radius, decay_function)

            #Adapt learning rate to each neuron based on position
            bmu_matrix = tf.stack([bmu_location for i in range(x*y)])
            bmu_distance = tf.reduce_sum(tf.pow(tf.subtract(self._locations, bmu_matrix), 2), 1)
            neighbourhood_func = tf.exp(tf.negative(tf.div(tf.cast(bmu_distance, "float32"),
tf.pow(_current_radius, 2))))
            learning_rate_matrix = tf.multiply(_current_learning_rate, neighbourhood_func)
```

```
        #Update all the weights
        multiplytiplier = tf.stack([tf.tile(tf.slice(
            learning_rate_matrix, np.array([i]), np.array([1])), [input_dim])
                                    for i in range(x*y)])
        delta = tf.multiply(
            multiplytiplier,
            tf.subtract(tf.stack([self._input for i in range(x*y)]), self._weights))

        new_weights = tf.add(self._weights, delta)
        self._training = tf.assign(self._weights, new_weights)

        #Initilize session and run it
        self._sess = tf.Session()
        initialization = tf.global_variables_initializer()
        self._sess.run(initialization)

    def train(self, input_vects):
        for iter_no in range(self._num_iter):
            for input_vect in input_vects:
                self._sess.run(self._training,
                               feed_dict={self._input: input_vect,
                                          self._iter_input: iter_no})

        self._centroid_matrix = [[] for i in range(self._x)]
        self._weights_list = list(self._sess.run(self._weights))
        self._locations = list(self._sess.run(self._locations))
        for i, loc in enumerate(self._locations):
            self._centroid_matrix[loc[0]].append(self._weights_list[i])

    def map_input(self, input_vectors):
        return_value = []
        for vect in input_vectors:
            min_index = min([i for i in range(len(self._weights_list))],
```

```
                key=lambda x: np.linalg.norm(vect – self._weights_list[x]))
            return_value.append(self._locations[min_index])
        return return_value

    def _generate_index_matrix(self, x,y):
        return tf.constant(np.array(list(self._iterator(x, y))))

    def _iterator(self, x, y):
        for i in range(x):
            for j in range(y):
                yield np.array([i, j])
```

That is quite a lot of code, so let's dissect it into smaller chunks and explain what each piece means. The majority of the code is in the constructor of class which, similar to the MiniSOM implementation, takes dimensions of the Self-Organizing Map, input dimensions, radius, and learning rate as input parameters.

8.3.2 Initialization

The first thing that is done is the initialization of all the fields with the values that are passed into the class constructor:

```
##Initialize properties
self._x = x
self._y = y
self._learning_rate = float(learning_rate)
self._radius = float(radius)
self._num_iter = num_iter
self._graph = tf.Graph()
```

Note that we created the *TensorFlow* graph as a *_graph* field. In the next part of the code, we essentially add operations to this graph and initialize our Self-Organizing Map. If you need more information on how *TensorFlows* graphs and sessions work, you can find it here. Anyway, the first step that needs to be done is to initialize variables and placeholders:

```
#Initializing variables and placeholders
self._weights = tf.Variable(tf.random_normal([x*y, input_dim]))
```

```
self._locations = self._generate_index_matrix(x, y)
self._input = tf.placeholder("float", [input_dim])
self._iter_input = tf.placeholder("float")
```

Basically, we created *_weights* as a randomly initialized tensor. In order to easily manipulate the neurons matrix of indexes is created – *_locations*. They are generated by using *_generate_index_matrix*, which looks like this:

```
def _generate_index_matrix(self, x,y):
    return tf.constant(np.array(list(self._iterator(x, y))))

def _iterator(self, x, y):
for i in range(x):
    for j in range(y):
        yield np.array([i, j])
```

Also, notice that *_input* (input vector) and *_iter_input* (iteration number, which is used for radius calculations) are defined as placeholders. This is since this information is filled during the training phase, not the construction phase. Once all variables and placeholders are initialized, we can start with the Self-Organizing Map learning process algorithm.

8.3.3 BMU Calculations

Firstly, BMU is calculated and it's location is determined:

```
#Calculating BMU
input_matix = tf.stack([self._input for i in range(x*y)])
distances = tf.sqrt(tf.reduce_sum(tf.pow(tf.subtract(self._weights, input_matix), 2), 1))
bmu = tf.argmin(distances, 0)


#Get BMU location
mask = tf.pad(tf.reshape(bmu, [1]), np.array([[0, 1]]))
size = tf.cast(tf.constant(np.array([1, 2])), dtype=tf.int64)
bmu_location = tf.reshape(tf.slice(self._locations, mask, size), [2])
```

The first part calculates the Euclidean distances between all neurons and the input vector. Don't get confused by the first line of this code. In essence, this input sample vector is repeated and the matrix is created to be used for calculations with weights tensor. Once distances are calculated, the index of the BMU is returned. This index is used, in the second part of the gist, to get the BMU location. We relied on the *slice* function for this. Once that is done, we need to calculate values for learning rate and radius for the current iteration. That is done like this:

```
#Adapt learning rate to each neuron based on position
bmu_matrix = tf.stack([bmu_location for i in range(x*y)])
bmu_distance = tf.reduce_sum(tf.pow(tf.subtract(self._locations, bmu_matrix), 2), 1)
neighbourhood_func = tf.exp(tf.negative(tf.div(tf.cast(bmu_distance, "float32"),
tf.pow(_current_radius, 2))))
learning_rate_matrix = tf.multiply(_current_learning_rate, neighbourhood_func)
```

The first matrix of BMU location value is created. Then of the neuron to the BMU is calculated. After that, the so-called *neighbourhood_func* is created. This function is basically defining how the weight of concrete neurons will be changed.

8.3.4 Update Weights

Finally, the weights are updated accordingly and the *TensorFlow* session is initialized and run:

```
#Update all the weights
multiplytiplier = tf.stack([tf.tile(tf.slice(
   learning_rate_matrix, np.array([i]), np.array([1])), [input_dim])
                        for i in range(x*y)])
delta = tf.multiply(
   multiplytiplier,
   tf.subtract(tf.stack([self._input for i in range(x*y)]), self._weights))

new_weightages = tf.add(self._weights, delta)
self._training = tf.assign(self._weights, new_weightages)

#Initilize session and run it
self._sess = tf.Session()
```

```
initialization = tf.global_variables_initializer()
self._sess.run(initialization)
```

Apart from the *_generate_index_matrix* function that you saw previously, this class has also two important functions – *train* and *map_input.* The first one, as its name suggests, is used to train the Self-Organizing Map with proper input. Here is how that function looks like:

```
def train(self, input_vects):
    for iter_no in range(self._num_iter):
        for input_vect in input_vects:
            self._sess.run(self._training,
                           feed_dict={self._input: input_vect,
                                      self._iter_input: iter_no})

    self._centroid_matrix = [[] for i in range(self._x)]
    self._weights_list = list(self._sess.run(self._weights))
    self._locations = list(self._sess.run(self._locations))
    for i, loc in enumerate(self._locations):
        self._centroid_matrix[loc[0]].append(self._weights_list[i])
```

Essentially, we have just run a defined number of iterations on passed input data. For that, we used _training operation that we created during class construction. Notice that here placeholders for iteration number and input sample are filled. That is how we run created sessions with correct data.

The second function that this class has is map_input. This function is mapping defined input samples to the correct output. Here is how it looks like:

```
def map_input(self, input_vectors):
    return_value = []
    for vect in input_vectors:
        min_index = min([i for i in range(len(self._weights_list))],
                        key=lambda x: np.linalg.norm(vect – self._weights_list[x]))
        return_value.append(self._locations[min_index])
    return return_value
```

8.3.5 Usage

In the end, we got a Self-Organizing Map with a pretty straightforward API that can be easily used. In the next article, we will use this class to solve one real-world problem. To sum it up, it can be used something like this:

```
from somtf import SOM

som = SOM(6, 6, 4, 0.5, 0.5, 100)
som.train(data)
```

8.4 ART INTRODUCTION

This network was developed by Stephen Grossberg and Gail Carpenter in 1987. It is based on competition and uses an unsupervised learning model. Adaptive Resonance Theory .ART networks, as the name suggests, is always open to new learning adaptive without losing the old patterns resonance. Basically, ART network is a vector classifier which accepts an input vector and classifies it into one of the categories depending upon which of the stored pattern it resembles the most.

8.4.1 Operating Principal

The main operation of ART classification can be divided into the following phases −

- **Recognition phase** − The input vector is compared with the classification presented at every node in the output layer. The output of the neuron becomes “1” if it best matches with the classification applied, otherwise it becomes “0”.
- **Comparison phase** − In this phase, a comparison of the input vector to the comparison layer vector is done. The condition for reset is that the degree of similarity would be less than vigilance parameter.
- **Search phase** − In this phase, the network will search for reset as well as the match done in the above phases. Hence, if there would be no reset and the match is quite good, then the classification is over. Otherwise, the process would be repeated and the other stored pattern must be sent to find the correct match.

8.5 ART1

It is a type of ART, which is designed to cluster binary vectors. We can understand about this with the architecture of it.

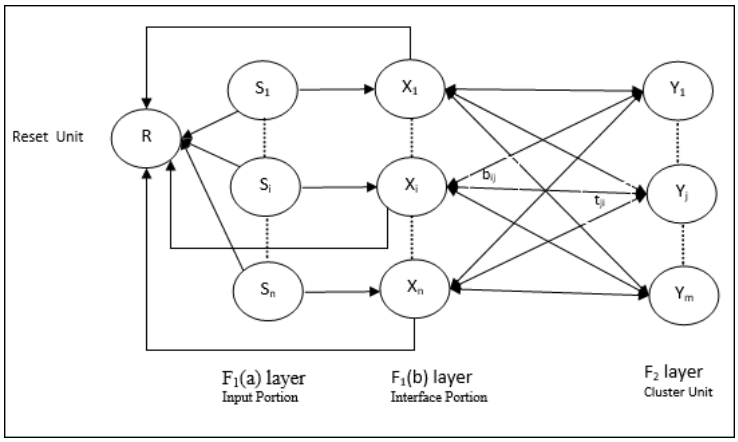
Architecture of ART1

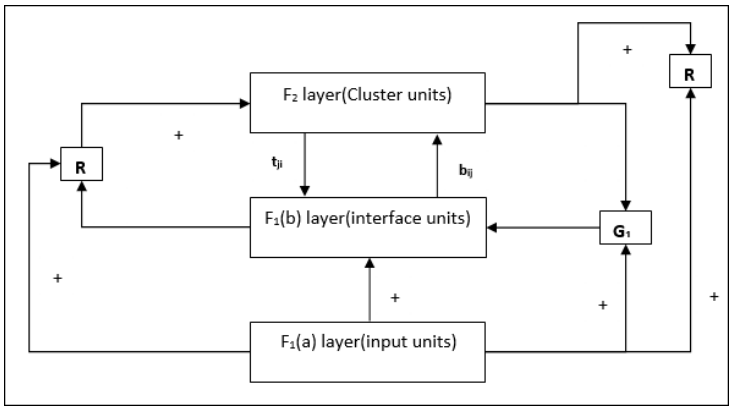
It consists of the following two units −

Computational Unit − It is made up of the following −

- **Input unit (F_1 layer)** − It further has the following two portions −
    - **F_1aa layer** InputportionInputportion − In ART1, there would be no processing in this portion rather than having the input vectors only. It is connected to F_1bb layer interfaceportioninterfaceportion.
    - **F_1bb layer** InterfaceportionInterfaceportion − This portion combines the signal from the input portion with that of F_2 layer. F_1bb layer is connected to F_2 layer through bottom up weights $\mathbf{b_{ij}}$ and F_2 layer is connected to F_1bb layer through top down weights $\mathbf{t_{ji}}$.
- **Cluster Unit (F_2 layer)** − This is a competitive layer. The unit having the largest net input is selected to learn the input pattern. The activation of all other cluster unit are set to 0.
- **Reset Mechanism** − The work of this mechanism is based upon the similarity between the top-down weight and the input vector. Now, if the degree of this similarity is less than the vigilance parameter, then the cluster is not allowed to learn the pattern and a rest would happen.

Supplement Unit − Actually the issue with Reset mechanism is that the layer $\mathbf{F_2}$ must have to be inhibited under certain conditions and must also be available when some learning happens. That is why two supplemental units namely, $\mathbf{G_1}$ and $\mathbf{G_2}$ is added along with reset unit, **R**. They are called **gain control units**. These units receive and send signals to the other units present in the network. **'+'** indicates an excitatory signal, while **'–'** indicates an inhibitory signal.





Parameters Used

Following parameters are used −

- **n** − Number of components in the input vector
- **m** − Maximum number of clusters that can be formed
- **b_{ij}** − Weight from F_1bb to F_2 layer, i.e. bottom-up weights
- **t_{ji}** − Weight from F_2 to F_1bb layer, i.e. top-down weights
- **ρ** − Vigilance parameter
- **||x||** − Norm of vector x

8.6 ALGORITHM

Step 1 − Initialize the learning rate, the vigilance parameter, and the weights as follows −

$\alpha>1 and 0<\rho\leq 1 \alpha>1 and 0<\rho\leq 1$

0<bij(0)<αα−1+nandtij(0)=10<bij(0)<αα−1+nandtij(0)=1

Step 2 − Continue step 3-9, when the stopping condition is not true.

Step 3 − Continue step 4-6 for every training input.

Step 4 − Set activations of all F_1aa and F_1 units as follows

F_2 = 0 and F_1aa = input vectors

Step 5 − Input signal from F_1aa to F_1bb layer must be sent like

si=xisi=xi

Step 6 − For every inhibited F_2 node

yj=∑ibijxiyj=∑ibijxi the condition is $\mathbf{y_j \neq -1}$

Step 7 − Perform step 8-10, when the reset is true.

Step 8 − Find **J** for $\mathbf{y_J \geq y_j}$ for all nodes **j**

Step 9 − Again calculate the activation on F_1bb as follows

xi=sitJixi=sitJi

Step 10 − Now, after calculating the norm of vector **x** and vector **s**, we need to check the reset condition as follows −

If **||x||/ ||s||** < vigilance parameter **ρ**, then inhibit node **J** and go to step 7

Else If **||x||/ ||s||** ≥ vigilance parameter **ρ**, then proceed further.

Step 11 − Weight updating for node **J** can be done as follows −

bij(new)=αxiα−1+||x||bij(new)=αxiα−1+||x||

tij(new)=xitij(new)=xi

Step 12 − The stopping condition for algorithm must be checked and it may be as follows −

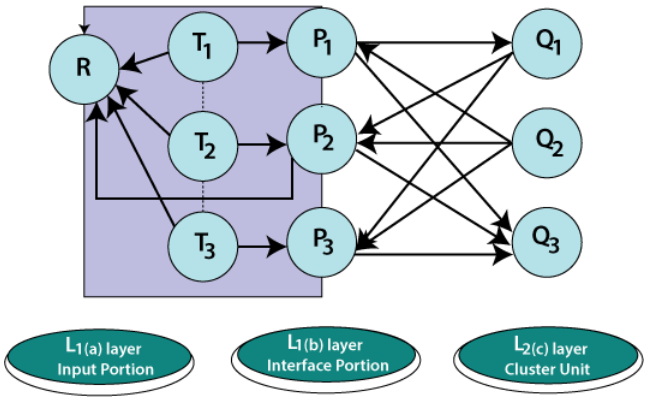
- Do not have any change in weight.
- Reset is not performed for units.
- Maximum number of epochs reached.

The Adaptive Resonance Theory (ART) was incorporated as a hypothesis for human cognitive data handling. The hypothesis has prompted neural models for pattern recognition and unsupervised learning. ART system has been utilized to clarify different types of cognitive and brain data.

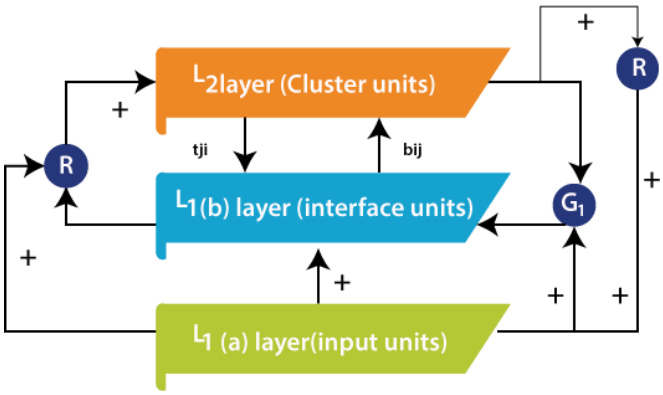
The Adaptive Resonance Theory addresses the **stability-plasticity**(stability can be defined as the nature of memorizing the learning and plasticity refers to the fact that they are flexible to gain new information) dilemma of a system that asks how learning can proceed in response to huge input patterns and simultaneously not to lose the stability for irrelevant patterns. Other than that, the stability-elasticity dilemma is concerned about how a system can adapt new data while keeping what was learned before. For such a task, a feedback mechanism is included among the ART neural network layers. In this neural network, the data in the form of processing elements output reflects back and ahead among layers. If an appropriate pattern is build-up, the resonance is reached, then adaption can occur during this period.

It can be defined as the formal analysis of how to overcome the learning instability accomplished by a competitive learning model, let to the presentation of an expended hypothesis, called **adaptive resonance theory** (ART). This formal investigation indicated that a specific type of top-down learned feedback and matching mechanism could significantly overcome the instability issue. It was understood that top-down attentional mechanisms, which had prior been found through an investigation of connections among cognitive and reinforcement mechanisms, had similar characteristics as these code-stabilizing mechanisms. In other words, once it was perceived how to solve the instability issue formally, it also turned out to be certain that one did not need to develop any quantitatively new mechanism to do so. One only needed to make sure to incorporate previously discovered attentional mechanisms. These additional mechanisms empower code learning to self- stabilize in response to an essentially arbitrary input system. **Grossberg** presented the basic principles of the adaptive resonance theory. A category of ART called ART1 has been described as an arrangement of ordinary differential equations by carpenter and Grossberg. These theorems can predict both the order of search as the function of the learning history of the system and the input patterns.

ART1 is an unsupervised learning model primarily designed for recognizing binary patterns. It comprises an attentional subsystem, an orienting subsystem, a vigilance parameter, and a reset module, as given in the figure given below. The vigilance parameter has a huge effect on the system. High vigilance produces higher detailed memories. The ART1 attentional comprises of two competitive networks, comparison field layer L1 and the recognition field layer L2, two control gains, Gain1 and Gain2, and two short-term memory (STM) stages S1 and S2. Long term memory (LTM) follows somewhere in the range of S1 and S2 multiply the signal in these pathways.



- Gains control empowers L1 and L2 to recognize the current stages of the running cycle. STM reset wave prevents active L2 cells when mismatches between bottom-up and top-down signals happen at L1. The comparison layer gets the binary external input passing it to the recognition layer liable for coordinating it to a classification category. This outcome is given back to the comparison layer to find out when the category coordinates the input vector. If there is a match, then a new input vector is read, and the cycle begins once again. If there is a mismatch, then the orienting system is in charge of preventing the previous category from getting a new category match in the recognition layer. The given two gains control the activity of the recognition and the comparison layer, respectively. The reset wave specifically and enduringly prevents active L2 cell until the current is stopped. The offset of the input pattern ends its processing L1 and triggers the offset of Gain2. Gain2 offset causes consistent decay of STM at L2 and thereby prepares L2 to encode the next input pattern without bais.



8.7 ART1 IMPLEMENTATION PROCESS

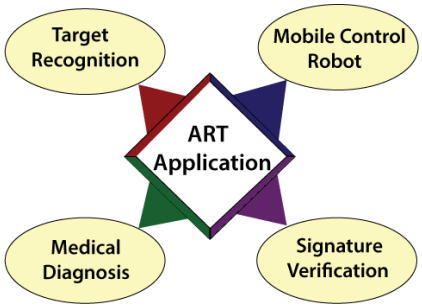
ART1 is a self-organizing neural network having input and output neurons mutually couple using bottom-up and top-down adaptive weights that perform recognition. To start our methodology, the system is first trained as per the adaptive resonance theory by inputting reference pattern data under the type of 5*5 matrix into the neurons for clustering within the output neurons. Next, the maximum number of nodes in L2 is defined following by the vigilance parameter. The inputted pattern enrolled itself as short term memory activity over a field of nodes L1. Combining and separating pathways from L1 to coding field L2, each weighted by an adaptive long-term memory track, transform into a net signal vector T. Internal competitive dynamics at L2 further transform T, creating a compressed code or content addressable memory. With strong competition, activation is concentrated at the L2 node that gets the maximal L1 → L2 signal. The primary objective of this work is divided into four phases as follows Comparision, recognition, search, and learning.

Advantage of adaptive learning theory(ART):

- It can be coordinated and utilized with different techniques to give more precise outcomes.
- It doesn't ensure stability in forming clusters.
- It can be used in different fields such as face recognition, embedded system, and robotics, target recognition, medical diagnosis, signature verification, etc.
- It shows stability and is not disturbed by a wide range of inputs provided to inputs.
- It has got benefits over competitive learning. The competitive learning cant include new clusters when considered necessary.

Application of ART:

- ART stands for Adaptive Resonance Theory. ART neural networks used for fast, stable learning and prediction have been applied in different areas. The application incorporates target recognition, face recognition, medical diagnosis, signature verification, mobile control robot.



- **Target recognition:**

  Fuzzy ARTMAP neural network can be used for automatic classification of targets depend on their radar range profiles. Tests on synthetic data show the fuzzy ARTMAP can result in substantial savings in memory requirements when related to k nearest neighbor(kNN) classifiers. The utilization of multiwavelength profiles mainly improves the performance of both kinds of classifiers.

- **Medical diagnosis:**

  Medical databases present huge numbers of challenges found in general information management settings where speed, use, efficiency, and accuracy are the prime concerns. A direct objective of improved computer-assisted medicine is to help to deliver intensive care in situations that may be less than ideal. Working with these issues has stimulated several ART architecture developments, including ARTMAP-IC.

- **Signature verification:**

  Automatic signature verification is a well known and active area of research with various applications such as bank check confirmation, ATM access, etc. the training of the

network is finished using ART1 that uses global features as input vector and the verification and recognition phase uses a two-step process. In the initial step, the input vector is coordinated with the stored reference vector, which was used as a training set, and in the second step, cluster formation takes place.

- **Mobile control robot:**

  Nowadays, we perceive a wide range of robotic devices. It is still a field of research in their program part, called artificial intelligence. The human brain is an interesting subject as a model for such an intelligent system. Inspired by the structure of the human brain, an artificial neural emerges. Similar to the brain, the artificial neural network contains numerous simple computational units, neurons that are interconnected mutually to allow the transfer of the signal from the neurons to neurons. Artificial neural networks are used to solve different issues with good outcomes compared to other decision algorithms.

  Limitations of ART:

- Some ART networks are contradictory as they rely on the order of the training data, or upon the learning rate.

ART implemntation with specification is described below

```
#!/usr/bin/env
python
        # ----------------------------------------------------------------------------
        # Adaptive Resonance Theory

        #
        # Distributed under the terms of the BSD License.
        # ----------------------------------------------------------------------------
        # Reference: Grossberg, S. (1987)
        #       Competitive learning: From interactive activation to
        #       adaptive resonance, Cognitive Science, 11, 23-63
```

```
#
# Requirements: python 2.5 or above => http://www.python.org
#               numpy  1.0 or above => http://numpy.scipy.org
# ------------------------------------------------------------------------------
from __future__import print_function
from __future__import division
import numpy as np
class ART:
    ''' ART class
    Usage example:
     ----------------
    # Create a ART network with input of size 5 and 20 internal units
    >>> network = ART(5,10,0.5)
    '''



    def __init__(self, n=5, m=10, rho=.5):
        '''
        Create network with specified shape
        Parameters:
         -------------
        n : int
            Size of input
        m : int
            Maximum number of internal units
        rho : float
            Vigilance parameter
        '''
        # Comparison layer
        self.F1 = np.ones(n)
        # Recognition layer
```

```
    self.F2 = np.ones(m)
    # Feed-forward weights
    self.Wf = np.random.random((m,n))
    # Feed-back weights
    self.Wb = np.random.random((n,m))
    # Vigilance
    self.rho = rho
    # Number of active units in F2
    self.active = 0




def learn(self, X):
    ''' Learn X '''


    # Compute F2 output and sort them (I)
    self.F2[...] = np.dot(self.Wf, X)
    I = np.argsort(self.F2[:self.active].ravel())[::-1]


    for i in I:
        # Check if nearest memory is above the vigilance level
        d = (self.Wb[:,i]*X).sum()/X.sum()
        if d >= self.rho:
            # Learn data
            self.Wb[:,i] *= X
             self.Wf[i,:] = self.Wb[:,i]/(0.5+self.Wb[:,i].sum())
            return self.Wb[:,i], i
```

```
        # No match found, increase the number of active units
        # and make the newly active unit to learn data
        if self.active < self.F2.size:
            i = self.active
            self.Wb[:,i] *= X
             self.Wf[i,:] = self.Wb[:,i]/(0.5+self.Wb[:,i].sum())
            self.active += 1
            return self.Wb[:,i], i


        return None,None




# --------------------------------------------------------------------------------
if __name__== '__main__':


    np.random.seed(1)


    # Example 1 : very simple data
    # --------------------------------------------------------------------------------
    network = ART( 5, 10, rho=0.5)
    data = ["   O ",
            "  O O",
            "    O",
            "  O O",
            "    O",
```

```
        "  O O",
        "    O",
        " OO O",
        " OO  ",
        " OO O",
        " OO  ",
        "OOO  ",
        "OO   ",
        "O    ",
        "OO   ",
        "OOO  ",
        "OOOO ",
        "OOOOO",
        "O    ",
        " O   ",
        "  O  ",
        "   O ",
        "    O",
        "  O O",
        " OO O",
        " OO  ",
        "OOO  ",
        "OO   ",
        "OOOO ",
        "OOOOO"]
X = np.zeros(len(data[0]))
for i in range(len(data)):
    for j in range(len(data[i])):
        X[j] = (data[i][j] == 'O')
    Z, k = network.learn(X)
    print("|%s|"%data[i],"-> class", k)
```

```
# Example 2 : Learning letters
# ------------------------------------------------------------------------------
def letter_to_array(letter):
    ''' Convert a letter to a numpy array '''
    shape = len(letter), len(letter[0])
    Z = np.zeros(shape, dtype=int)
    for row in range(Z.shape[0]):
        for column in range(Z.shape[1]):
            if letter[row][column] == '#':
                Z[row][column] = 1
    return Z


def print_letter(Z):
    ''' Print an array as if it was a letter'''
    for row in range(Z.shape[0]):
        for col in range(Z.shape[1]):
            if Z[row,col]:
                print( '#', end="" )
            else:
                print( ' ', end="" )
        print( )

A = letter_to_array( [' #### ',
                      '#    #',
                      '#    #',
                      '######',
```

```
                '#    #',
                '#    #',
                '#    #'] )
B = letter_to_array( ['##### ',
                '#    #',
                '#    #',
                '##### ',
                '#    #',
                '#    #',
                '##### '] )
C = letter_to_array( [' #### ',
                '#    #',
                '#     ',
                '#     ',
                '#     ',
                '#    #',
                ' #### '] )
D = letter_to_array( ['##### ',
                '#    #',
                '#    #',
                '#    #',
                '#    #',
                '#    #',
                '##### '] )
E = letter_to_array( ['######',
                '#     ',
                '#     ',
                '####  ',
                '#     ',
                '#     ',
                '######'] )
```

```
F = letter_to_array( ['######',
                      '#     ',
                      '#     ',
                      '####  ',
                      '#     ',
                      '#     ',
                      '#     '] )


samples = [A,B,C,D,E,F]
network = ART( 6*7, 10, rho=0.15 )


for i in range(len(samples)):
    Z, k = network.learn(samples[i].ravel())
    print("%c"%(ord('A')+i),"-> class",k)
    print_letter(Z.reshape(7,6))
```