BACHELOR OF TECHNOLOGY

IN

CHEMICAL ENGINEERING

PRACTICAL FILE

OF

MACHINE LEARNING

(AI&ML SPECIALIZATION)

SUBMITTED TO:

Mrs. Sunita Dahiya

ECE Department

)

SUBMITTED BY:

Name :- Mayank Sharma

CLASS:-B.TECH(CHE 3RD YEAR

ROLL NO-21001005016

INDEX

S.NO	EXPERIMENT NAME	D.O.P	D.O.C
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

EXPERIMENT:1

AIM: To Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file.

THEORY: The find-S algorithm is a basic concept learning algorithm in machine learning. The find-S algorithm finds the most specific hypothesis that fits all the positive examples. We have to note here that the algorithm considers only those positive training example. The find-S algorithm starts with the most specific hypothesis and generalizes this hypothesis each time it fails to classify an observed positive training data

What is Find-S Algorithm in Machine Learning?

In order to understand Find-S algorithm, you need to have a basic idea of the following concepts as well:
Concept Learning
General Hypothesis
Specific Hypothesis

1. Concept Learning

Let's try to understand concept learning with a real-life example. Most of human learning is based on past instances or experiences. For example, we are able to identify any type of vehicle based on a certain set of features like make, model, etc., that are defined over a large set of features.

These special features differentiate the set of cars, trucks, etc from the larger set of vehicles. These features that define the set of cars, trucks, etc are known as concepts.

Similar to this, machines can also learn from concepts to identify whether an object belongs to a specific category or not. Any algorithm that supports concept learning requires the following:

Training Data

Target Concept

Actual Data Objects

2. General Hypothesis

Hypothesis, in general, is an explanation for something. The general hypothesis basically states the general relationship between the major variables. For example, a general hypothesis for ordering food would be *I want a burger.*

G = { '?', '?', '?', …..'?'}

3. Specific Hypothesis

The specific hypothesis fills in all the important details about the variables given in the general hypothesis. The more specific details into the example given above would be I want a cheeseburger with a chicken pepperoni filling with a lot of lettuce.

S = {'Φ','Φ','Φ', ……,'Φ'}

Important Representation :

1. ? indicates that any value is acceptable for the attribute.
2. specify a single required value (e.g., Cold) for the attribute.
3. ϕindicates that no value is acceptable.
4. The most general hypothesis is represented by: {?, ?, ?, ?, ?, ?}
5. The most specific hypothesis is represented by: {ϕ, ϕ, ϕ, ϕ, ϕ, ϕ}

Steps Involved In Find-S :

1. Start with the most specific hypothesis.
   h = {ϕ, ϕ, ϕ, ϕ, ϕ, ϕ}
2. Take the next example and if it is negative, then no changes occur to the hypothesis.
3. If the example is positive and we find that our initial hypothesis is too specific then we update our current hypothesis to a general condition.
4. Keep repeating the above steps till all the training examples are complete.
5. After we have completed all the training examples we will have the final hypothesis when can use to classify the new examples.

ALGORITHM:

1. Initialize h to the most specific hypothesis in H

2. For each positive training instance x

   For each attribute constraint a, in h

      If the constraint a, is satisfied by x

      Then do nothing

      Else replace a, in h by the next more general constraint that is satisfied by x

3. Output hypothesis h

CODE:

INPUT:

```
 #to read data in CSV file
data = pd.read_csv("/content/play.csv")
print(data)

            Time Weather Temperature Company Humidity    Wind Goes
    0  Morning   Sunny       Warm     Yes    Mild  Strong  Yes
    1  Evening   Rainy       Cold      No    Mild  Normal   No
    2  Morning   Sunny   Moderate     Yes  Normal  Normal  Yes
    3  Evening   Sunny       Cold     Yes    High  Strong  Yes

 [] #making an array of all attributes
d = np.array(data)[:,:-1]
print("The attributes are: ", d)
       The attributes are:  [['Morning' 'Sunny' 'Warm' 'Yes' 'Mild' 'Strong']
      ['Evening' 'Rainy' 'Cold' 'No' 'Mild' 'Normal']
      ['Morning' 'Sunny' 'Moderate' 'Yes' 'Normal' 'Normal']
      ['Evening' 'Sunny' 'Cold' 'Yes' 'High' 'Strong']]
)
 [] #segregating target that has positive and negative examples
target = np.array(data)[:,-1]
print("The target is : ", target


      The target is :  ['Yes' 'No' 'Yes' 'Yes']


 [ ]#Training function to implement find-s algo
def train(c,t):
  for i,val in enumerate(t):
    if val == "Yes":
     specific_hypothesis = c[i].copy()
     break
  for i,val in enumerate(c):
    if t[i] == "Yes":
      for x in range(len(specific_hypothesis)):
        if val[x] != specific_hypothesis[x]:
          specific_hypothesis[x] = '?'
        else:
            pass
  return specific_hypothesis
 [] #Final hypothesis
print("The final hypothesis is:",train(d,target))
```

RESULT: The final hypothesis is: ['?' 'Sunny' '?' 'Yes' '?' '?']

Mayank(21001005016)

EXPERIMENT:2

AIM: For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.

THEORY:

The candidate-elimination algorithm manipulates the boundary-set representation of a version space to create boundary sets that represent a new version space consistent with all the previous instances plus the new one.In case of positive examples, the algorithm generalizes the elements of the [sbs] as little as possible so that they cover the new instance yet remain consistent with past data, and removes those elements of the [gbs] that do not cover the new instance. And for a negative instance the algorithm specializes elements of the [gbs] so that they no longer cover the new instance yet remain consistent with past data, and removes from the [sbs] those elements that mistakenly cover the new, negative instance.

A hypothesis is sufficient if it is 1 for for all training samples labelled 1 and is said to be necessary if it is 0 for all training samples labelled 0. A hypothesis that is both necessary and sufficient is said to be consistent with our dataset.

Terms Used:

Concept learning: Concept learning is basically the learning task of the machine (Learn by Train data)

General Hypothesis: Not Specifying features to learn the machine.

G = {'?', '?','?','?'…}: Number of attributes

Specific Hypothesis: Specifying features to learn machine (Specific feature)

S= {'pi','pi','pi'…}: The number of pi depends on a number of attributes.

Version Space: It is an intermediate of general hypothesis and Specific hypothesis. It not only just writes one hypothesis but a set of all possible hypotheses based on training data-set.

Algorithm:

Step1: Load Data set
Step2: Initialize General Hypothesis and Specific Hypothesis.
Step3: For each training example
Step4: If example is positive example
    if attribute_value == hypothesis_value:

Do nothing
    else:
      replace attribute value with '?' (Basically generalizing it)

Step5: If example is Negative example
    Make generalize hypothesis more specific.

CODE:

```
import numpy as np
import pandas as pd

# Loading Data from a CSV File
data = pd.DataFrame(data=pd.read_csv('trainingdata.csv'))
print(data)

       sky airTemp humidity    wind water forecast enjoySport
    0  Sunny    Warm   Normal  Strong  Warm     Same       Yes
    1  Sunny    Warm     High  Strong  Warm     Same       Yes
    2  Rainy    Cold     High  Strong  Warm   Change        No
    3  Sunny    Warm     High  Strong  Cool   Change       Yes

# Separating concept features from Target
concepts = np.array(data.iloc[:,0:-1])
print(concepts)
   [['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
   ['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
   ['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
   ['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']]

# Isolating target into a separate DataFrame
# copying last column to target array
target = np.array(data.iloc[:,-1])
print(target)
     ['Yes' 'Yes' 'No' 'Yes']

def learn(concepts, target):

  '''
  learn() function implements the learning method of the Candidate elimination algorithm.
  Arguments:
     concepts - a data frame with all the features
     target - a data frame with corresponding output values
  '''
```

```
    # Initialise S0 with the first instance from concepts
    # .copy() makes sure a new list is created instead of just pointing to the same memory
location
    specific_h = concepts[0].copy()
    print("\nInitialization of specific_h and general_h")
    print(specific_h)
    #h=["#" for i in range(0,5)]
    #print(h)

    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print(general_h)
    # The learning iterations
    for i, h in enumerate(concepts):

        # Checking if the hypothesis has a positive target
        if target[i] == "Yes":
            for x in range(len(specific_h)):

                # Change values in S & G only if values change
                if h[x] != specific_h[x]:
                    specific_h[x] = '?'
                    general_h[x][x] = '?'

        # Checking if the hypothesis has a positive target
        if target[i] == "No":
            for x in range(len(specific_h)):
                # For negative hyposthesis change values only  in G
                if h[x] != specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = '?'

        print("\nSteps of Candidate Elimination Algorithm",i+1)
        print(specific_h)
        print(general_h)

    # find indices where we have empty rows, meaning those that are unchanged
    indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?']]
    for i in indices:
        # remove those rows from general_h
        general_h.remove(['?', '?', '?', '?', '?', '?'])
    # Return final values
    return specific_h, general_h
s_final, g_final = learn(concepts, target)
print("\nFinal Specific_h:", s_final, sep="\n")
```

print("\nFinal General_h:", g_final, sep="\n")
Initialization of specific_h and general_h
['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 1
['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
Steps of Candidate Elimination Algorithm 2
['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 3
['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]

Steps of Candidate Elimination Algorithm 4
['Sunny' 'Warm' '?' 'Strong' '?' '?']
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

RESULT:

Final Specific_h:
['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final General_h:
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]

EXPERIMENT:3

AIM: Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

THEORY:

A Decision Tree is a supervised Machine learning algorithms used for both regression and classification problem statement. It uses the tree representation to solve a problem in which each node represents an attribute, each link represents a decision rule and each leaf represents an outcome(categorical or continuous value).

Decision Tree Terminologies

Root Node- It is the topmost node in the tree, which represent the complete dataset. Also we can say it is the starting point of the decision-making process.

Decision/Internal Node- Decision nodes are nothing but the result in the splitting of data into multiple data segments and main goal is to have the children nodes with maximum homogeneity or purity(means all of the same kind).

Leaf/Terminal Node- This node represent the data section having highest homogeneity (means all of the same kind).

Entropy-It is used for checking the impurity or uncertainty present in the data. Entropy is used to evaluate the quality of a split. When entropy is zero the sample is completely homogeneous, meaning that each instance belongs to the same class and entropy is one when the sample is equally divided between different classes.

Information Gain- Information gain indicates how much information a particular feature/ variable give us about the final outcome.

ID3 [Iterative Dichotomiser3]

(It is the most popular algorithms used to constructing trees.)

ID3 stands for Iterative Dichotomizer3 and is named such because the algorithm iteratively(repeatedly) dichotomizes(divides) features into two or more groups at each step.ID3 is an algorithm invented by Ross Quinlan used to generate a decision tree from a dataset and is the most popular algorithms used to constructing trees.

ID3 is the core algorithm for building a decision tree .It employs a top-down greedy search through the space of all possible branches with no backtracking. This algorithm uses information gain and entropy to construct a classification decision tree.

Characteristics of ID3 Algorithm

Major Characteristics of the ID3 Algorithm are listed below:

ID3 can overfit the training data (to avoid overfitting, smaller decision trees should be preferred over larger ones).

This algorithm usually produces small trees, but it does not always produce the smallest possible tree.

ID3 is harder to use on continuous data (if the values of any given attribute is continuous, then there are many more places to split the data on this attribute, and searching for the best value to split by can be time-consuming).

Steps to making Decision Tree

a) Take the Entire dataset as an input.

b) Calculate the Entropy of the target variable, As well as the predictor attributes

c) Calculate the information gain of all attributes.

d) Choose the attribute with the highest information gain as the Root Node

e) Repeat the same procedure on every branch until the decision node of each branch is finalized.

CODE:

```
[ ]
#Importing the necessary basic python libraries
import pandas as pd #for manipulating the csv data
import numpy as np #for mathematical calculation
[ ]
#Reading dataset
train_data_m = pd.read_csv("/content/trainPlayTennis.csv") #importing the dataset from
the disk
train_data_m.head() #viewing some row of the dataset
```

	outlook	temp	humidity	windy	PlayTennis
0	sunny	hot	high	weak	no
1	sunny	hot	high	strong	no
2	overcast	hot	high	weak	yes
3	rainy	mild	high	weak	yes
4	rainy	cool	normal	weak	yes

Entropy of whole dataset:
Total row = 14, Row with "Yes" class = 9, Row with "No" class = 5, Complete entropy of dataset is: H(S) = - p(Yes) * log2(p(Yes)) - p(No) * log2(p(No))
= - (9/14) * log2(9/14) - (5/14) * log2(5/14) = - (-0.41) - (-0.53) = 0.94

[ ]

```
#Calculating the entropy for whole dataset
def calc_total_entropy(train_data, label, class_list):
   total_row = train_data.shape[0] #the total size of the dataset
   total_entr = 0

   for c in class_list: #for each class in the label
      total_class_count = train_data[train_data[label] == c].shape[0] #number of the class
      total_class_entr = -(total_class_count/total_row)*np.log2(total_class_count/total_row)
#entropy of the class
      total_entr += total_class_entr #adding the class entropy to the total entropy of the
dataset
   return total_entr
```

Entropy of filtered dataset
Categorical values of Outlook - Sunny, Overcast and Rain

Total count of row containing: Sunny = 5 Sunny & Yes = 2 Sunny & No = 3

H(Outlook=Sunny) = -(2/5)log(2/5)-(3/5)log(3/5) = 0.971

Total count of row containing: Rain = 5 Rain & Yes = 3 Rain & No = 2

H(Outlook=Rain) = -(3/5)log(3/5)-(2/5)log(2/5) = 0.971

Total count of row containing: Overcast = 4 Overcast & Yes = 4 Overcast & No = 0

H(Outlook=Overcast) = -(4/4)*log(4/4)-0 = 0

Similar exercise for all features like Wind, Humidity etc.

[ ]

```
#Calculating the entropy for the filtered dataset
def calc_entropy(feature_value_data, label, class_list):
```

```
    class_count = feature_value_data.shape[0]
    entropy = 0

    for c in class_list:
        label_class_count = feature_value_data[feature_value_data[label] == c].shape[0] #row
count of class c
        entropy_class = 0
        if label_class_count != 0:
            probability_class = label_class_count/class_count #probability of the class
            entropy_class = - probability_class * np.log2(probability_class)  #entropy
        entropy += entropy_class
    return entropy
```

After calculating entropy, calculate the information gain of that feature (e.g. for the feature Outlook):

Infirmation: I(Outlook) = p(Sunny) * H(Outlook=Sunny) + p(Rain) * H(Outlook=Rain) + p(Overcast) * H(Outlook=Overcast) = (5/14)0.971 + (5/14)0.971 + (4/14)*0 = 0.693

Information Gain = H(S) - I(Outlook) = 0.94 - 0.693 = 0.247

```
[ ]
#Calculating information gain for a feature
def calc_info_gain(feature_name, train_data, label, class_list):
    feature_value_list = train_data[feature_name].unique() #unqiue values of the feature
    total_row = train_data.shape[0]
    feature_info = 0.0
for feature_value in feature_value_list:
        feature_value_data = train_data[train_data[feature_name] == feature_value] #filtering
rows with that feature_value
        feature_value_count = feature_value_data.shape[0]
        feature_value_entropy = calc_entropy(feature_value_data, label, class_list)
#calculcating entropy for the feature value
        feature_value_probability = feature_value_count/total_row
        feature_info += feature_value_probability * feature_value_entropy #calculating
information of the feature value
    return calc_total_entropy(train_data, label, class_list) - feature_info #calculating
information gain by subtracting
```

Like Outlook feature, calculate information gain for every feature in the dataset and select the feature with the highest information gain.

Information gain: Outlook = 0.247 (Highest value)

Temperature = 0.0292

Humidity = 0.153

Wind = 0.048

[ ]
```
#Finding the most informative feature (feature with highest information gain)
def find_most_informative_feature(train_data, label, class_list):
   feature_list = train_data.columns.drop(label) #finding the feature names in the dataset
                                       #N.B. label is not a feature, so dropping it
   max_info_gain = -1
   max_info_feature = None

   for feature in feature_list:  #for each feature in the dataset
      feature_info_gain = calc_info_gain(feature, train_data, label, class_list)
      if max_info_gain < feature_info_gain: #selecting feature name with highest information
gain
         max_info_gain = feature_info_gain
         max_info_feature = feature

   return max_info_feature
```

Generate a node (feature having highest information gain) in the tree and its value as a branch. e.g. selected feature is Outlook, so add Outlook as a node in the tree and its value Sunny or Rain or Overcast as a branch.

Outlook is selected as Node.

(Outlook = Sunny): Not pure class, contains both class Yes and No
(Outlook = Overcast): Pure class, contains only one class Yes

(Outlook = Rain): Not pure class, contains both class Yes and No

After selecting a pure class, remove the rows from the dataset corresponding to the feature value. e.g. Outlook = Overcast.

This updated dataset will be used for the next iterations.

[ ]
```
#Adding a node to tree
def generate_sub_tree(feature_name, train_data, label, class_list):
   feature_value_count_dict = train_data[feature_name].value_counts(sort=False)
#dictionary of the count of unqiue feature value
   tree = {} #sub tree or node

   for feature_value, count in feature_value_count_dict.items():
```

```
        feature_value_data = train_data[train_data[feature_name] == feature_value] #dataset
with only feature_name = feature_value

        assigned_to_node = False #flag for tracking feature_value is pure class or not
        for c in class_list: #for each class
          class_count = feature_value_data[feature_value_data[label] == c].shape[0] #count of
class c
  if class_count == count: #count of (feature_value = count) of class (pure class)
            tree[feature_value] = c #adding node to the tree
            train_data = train_data[train_data[feature_name] != feature_value] #removing
rows with feature_value
            assigned_to_node = True
        if not assigned_to_node: #not pure class
          tree[feature_value] = "?" #as feature_value is not a pure class, it should be expanded
further,
                              #so the branch is marking with ?

    return tree, train_data

[ ]
#Performing ID3 Algorithm and generating Tree
def make_tree(root, prev_feature_value, train_data, label, class_list):
  if train_data.shape[0] != 0: #if dataset becomes enpty after updating
        max_info_feature = find_most_informative_feature(train_data, label, class_list) #most
informative feature
        tree, train_data = generate_sub_tree(max_info_feature, train_data, label, class_list)
#getting tree node and updated dataset
        next_root = None

        if prev_feature_value != None: #add to intermediate node of the tree
          root[prev_feature_value] = dict()
          root[prev_feature_value][max_info_feature] = tree
          next_root = root[prev_feature_value][max_info_feature]
        else: #add to root of the tree
          root[max_info_feature] = tree
          next_root = root[max_info_feature]

        for node, branch in list(next_root.items()): #iterating the tree node
          if branch == "?": #if it is expandable
            feature_value_data = train_data[train_data[max_info_feature] == node] #using the
updated dataset
            make_tree(next_root, node, feature_value_data, label, class_list) #recursive call
with updated dataset
```

[ ]
```
#Finding unique classes of the label and Starting the algorithm
def id3(train_data_m, label):
    train_data = train_data_m.copy() #getting a copy of the dataset
    tree = {} #tree which will be updated
    class_list = train_data[label].unique() #getting unqiue classes of the label
    make_tree(tree, None, train_data, label, class_list) #start calling recursion
    return tree
```

[ ]
```
tree = id3(train_data_m, 'PlayTennis')
print(tree)
```

{'outlook': {'sunny': {'humidity': {'high': 'no', 'normal': 'yes'}}, 'overcast': 'yes', 'rainy': {'windy': {'weak': 'yes', 'strong': 'no'}}}}

[ ]
```
def predict(tree, instance):
    if not isinstance(tree, dict): #if it is leaf node
        return tree #return the value
    else:
        root_node = next(iter(tree)) #getting first key/feature name of the dictionary
        feature_value = instance[root_node] #value of the feature
        if feature_value in tree[root_node]: #checking the feature value in current tree node
            return predict(tree[root_node][feature_value], instance) #goto next feature
        else:
            return None
```

[ ]
```
def evaluate(tree, test_data_m, label):
    correct_preditct = 0
    wrong_preditct = 0
    for index, row in test_data_m.iterrows(): #for each row in the dataset
        result = predict(tree, test_data_m.iloc[index]) #predict the row
   if result == test_data_m[label].iloc[index]: #predicted value and expected value is same or 
not
            correct_preditct += 1 #increase correct count
        else:
            wrong_preditct += 1 #increase incorrect count
    accuracy = correct_preditct / (correct_preditct + wrong_preditct) #calculating accuracy
    return accuracy
```

[ ]
test_data_m = pd.read_csv("/content/testPlayTennis.csv") #importing test dataset into dataframe

accuracy = evaluate(tree, test_data_m, 'PlayTennis') #evaluating the test dataset

print(accuracy)

RESULT:

Tree is: {'outlook': {'sunny': {'humidity': {'high': 'no', 'normal': 'yes'}}, 'overcast': 'yes', 'rainy': {'windy': {'weak': 'yes', 'strong': 'no'}}}}

Accuracy is: 1.0

Experiment:4

AIM: Implement Linear Regression for Salary prediction.

Theory:

Linear regression is one of the easiest and most popular Machine Learning algorithms. It is a statistical method that is used for predictive analysis. Linear regression makes predictions for continuous/real or numeric variables such as **sales, salary, age, product price,** etc.

Linear regression algorithm shows a linear relationship between a dependent (y) and one or more independent (y) variables, hence called as linear regression. Since linear regression shows the linear relationship, which means it finds how the value of the dependent variable is changing according to the value of the independent variable.

Mathematically, we can represent a linear regression as:

Y=a0+a1+E

Here,

Y= Dependent Variable (Target Variable)

X= Independent Variable (predictor Variable)

a0= intercept of the line (Gives an additional degree of freedom)

a1 = Linear regression coefficient (scale factor to each input value).

ε = random error

The values for x and y variables are training datasets for Linear Regression model representation.

Assumptions of Linear Regression

Below are some important assumptions of Linear Regression. These are some formal checks while building a Linear Regression model, which ensures to get the best possible result from the given dataset.

Linear relationship between the features and target:
Linear regression assumes the linear relationship between the dependent and independent variables.

Small or no multicollinearity between the features:
Multicollinearity means high-correlation between the independent variables. Due to multicollinearity, it may difficult to find the true relationship between the predictors and target variables. Or we can say , it is difficult to determine which predictor variable is

affecting the target variable and which is not. So, the model assumes either little or no multicollinearity between the features or independent variables.

Homoscedasticity Assumption:
Homoscedasticity is a situation when the error term is the same for all the values of independent variables. With homoscedasticity, there should be no clear pattern distribution of data in the scatter plot.

Normal distribution of error terms:
Linear regression assumes that the error term should follow the normal distribution pattern. If error terms are not normally distributed, then confidence intervals will become either too wide or too narrow , which may cause difficulties in finding coefficients.
It can be checked using the q-q plot. If the plot shows a straight line without any deviation, which means the error is normally distributed.

No autocorrelations:
The linear regression model assumes no autocorrelation in error terms. If there will be any correlation in the error term , then it will drastically reduce the accuracy of the model. Autocorrelation usually occurs if there is a dependency between residual errors.

Types of Linear Regression

Linear regression can be further divided into two types of the algorithm:

- **Simple Linear Regression:**

a single independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Simple Linear Regression.

- **Multiple Linear regression:**
  If more than one independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Multiple Linear Regression.

Code:

```
import pandas as pd
import matplotlib.pyplot as plt

[] data = pd.read_csv("/content/Salary_Data.csv")
  data.head()
```

	Years Experience	Salary
0	1.1	39343.0
1	1.3	46205.0
2	1.5	37731.0
3	2.0	43525.0
4	2.2	39891.0

```
[]data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30 entries, 0 to 29
Data columns (total 2 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   YearsExperience  30 non-null     float64
 1   Salary           30 non-null     float64
dtypes: float64(2)
memory usage: 608.0 bytes
```

```
[]### Divide the data into features and targets
 x = data.iloc[:,0].values
 y = data.iloc[:,1].values




print(x.shape)
print(y.shape)
```

```
   (30,)
   (30,)
```

```
[]x = x.reshape((30,1))
  x.shape
```

```
(30, 1)
```

```
### Split the data into training and testing samples
from sklearn.model_selection import train_test_split
xtrain,xtest,ytrain,ytest = train_test_split(x,y,train_size=0.80,random_state=200)
```

```
xtrain
```

```
xtest
```

```
#### Build the model
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

```
## Training the model
model.fit(xtrain,ytrain)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

```
### Prediction
ypred = model.predict(xtest)
ypred
```

```
array([108335.56338618, 125509.79795421,  39638.62511404, 100702.57024483,
        35822.12854337,  81620.08739146])
```

```
ytest
```

```
array([109431., 121872.,  37731., 101302.,  39343.,  81363.])
```

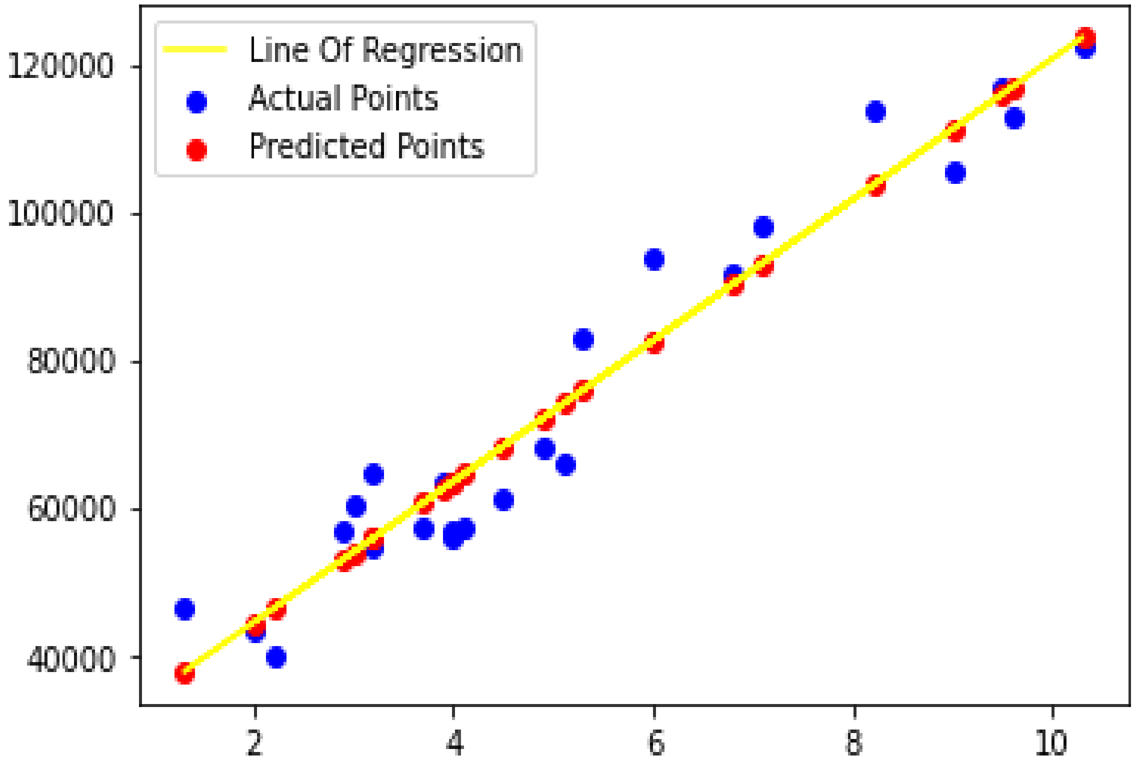
```
xtest
```

```
array([[ 8.7],
       [10.5],
       [ 1.5],
       [ 7.9],
       [ 1.1],
       [ 5.9]])
```

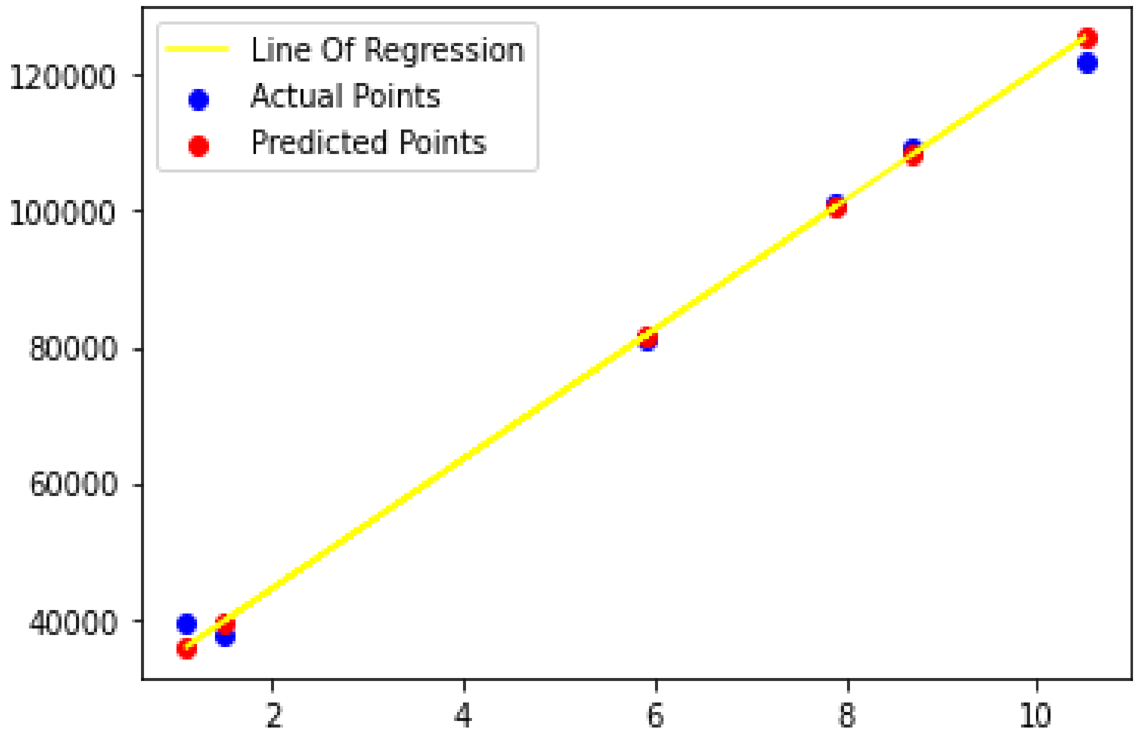
```
from sklearn.metrics import r2_score
score = r2_score(ytest,ypred)
score*100
```

99.52429110093546

```
### Draw the line of regression (Training samples)
plt.scatter(xtrain,ytrain,color='blue',label="Actual Points")
plt.scatter(xtrain,model.predict(xtrain),color='red',label="Predicted Points")
plt.plot(xtrain,model.predict(xtrain),color='yellow',label="Line Of Regression")
plt.legend()
plt.show()
```




```
### Draw the line of regression (Testing samples)
plt.scatter(xtest,ytest,color='blue',label="Actual Points")
plt.scatter(xtest,ypred,color='red' ,label="Predicted Points")
plt.plot(xtest,ypred,color='yellow' ,label="Line Of Regression")
plt.legend()
plt.show()
```




```
accuracy = []
for i in range(501):
  xtrain1,xtest1,ytrain1,ytest1 = train_test_split(x,y,train_size=0.80,random_state=i)
  model1 = LinearRegression()
  model1.fit(xtrain1,ytrain1)
  ypred1 = model1.predict(xtest1)
  score1 = r2_score(ytest1,ypred1)
  accuracy.append(score1)
```

```
accuracy
```

```
import numpy as np
np.max(accuracy)
```

0.9952429110093546

```
model.predict([[15]])[0]
```

168445.38437429562

```
model.predict([[15]])
```

array([168445.3843743])

Mayank(21001005016)

```
model.predict([[15]])
```
array([168445.3843743])

```
model.predict([[15]])
```
array([168445.3843743])

```
round(model.predict([[15]])[0],2)
```
168445.38

Result : Linear regression has been successfully implemented.