DEPARTMENT OF ARTIFICIAL INTELLIGENCE & DATA SCIENCE

LAB MANUAL

I SEMESTER

MACHINE LEARNING LAB

Prepared by

1.Dr. A S ANAKATH

2.MR.S.PONMANAM

NAME	
REG.NO.	
SUBJECT	

CONTENTS

CONTENTS

Introduction

Machine learning

Machine learning is a subset of artificial intelligence in the field of computer science that often uses statistical techniques to give computers the ability to "learn" (i.e., progressively improve performance on a specific task) with data, without being explicitly programmed. In the past decade, machine learning has given us self-driving cars, practical speech recognition, effective web search, and a vastly improved understanding of the human genome.

Machine learning tasks

Machine learning tasks are typically classified into two broad categories, depending on whether there is a learning "signal" or "feedback" available to a learning system:

1. **Supervised learning**: The computer is presented with example inputs and their desired outputs, given by a "teacher", and the goal is to learn a general rule that maps inputs to outputs. As special cases, the input signal can be only partially available, or restricted to special feedback:

2. **Semi-supervised learning**: the computer is given only an incomplete training signal: a training set with some (often many) of the target outputs missing.

3. **Active learning**: the computer can only obtain training labels for a limited set of instances (based on a budget), and also has to optimize its choice of objects to acquire labels for. When used interactively, these can be presented to the user for labeling.

4. **Reinforcement learning**: training data (in form of rewards and punishments) is given only as feedback to the program's actions in a dynamic environment, such as driving a vehicle or playing a game against an opponent.

5. **Unsupervised learning**: No labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end (feature learning).

Supervised learning	Un Supervised learning	Instance based learning
Find-s algorithm	EM algorithm	
Candidate elimination algorithm		
Decision tree algorithm		
Back propagation Algorithm		Locally weighted Regression algorithm
Naïve Bayes Algorithm	K means algorithm	
K nearest neighbour algorithm(lazy learning algorithm)		

Machine Learning Applications

In classification, inputs are divided into two or more classes, and the learner must produce a model that assigns unseen inputs to one or more (multi-label classification) of these classes. This is typically tackled in a supervised manner. Spam filtering is an example of classification, where the inputs are email (or other) messages and the classes are "spam" and "not spam".

In regression, also a supervised problem, the outputs are continuous rather than discrete.
In clustering, a set of inputs is to be divided into groups. Unlike in classification, the groups are not known beforehand, making this typically an unsupervised task.

Density estimation finds the distribution of inputs in some space.

Dimensionality reduction simplifies inputs by mapping them into a lower dimensional space. Topic modeling is a related problem, where a program is given a list of human language documents and is tasked with finding out which documents cover similar topics.

Machine learning Approaches

1. Decision tree learning

Decision tree learning uses a decision tree as a predictive model, which maps observations about an item to conclusions about the item's target value.

2. Association rule learning

Association rule learning is a method for discovering interesting relations between variables in large databases.

3. Artificial neural networks

An artificial neural network (ANN) learning algorithm, usually called "neural network" (NN), is a learning algorithm that is vaguely inspired by biological neural networks. Computations are structured in terms of an interconnected group of artificial neurons, processing information using a connectionist approach to computation. Modern neural networks are non-linear statistical data modeling tools. They are usually used to model complex relationships between inputs and outputs, to find patterns in data, or to capture the statistical structure in an unknown joint probability distribution between observed variables.

4. Deep learning

Falling hardware prices and the development of GPUs for personal use in the last few years have contributed to the development of the concept of deep learning which consists of multiple hidden layers in an artificial neural network. This approach tries to model the way the human brain processes light and sound into vision and hearing. Some successful applications of deep learning are computer vision and speech Recognition.

5. Inductive logic programming

Inductive logic programming (ILP) is an approach to rule learning using logic Programming as a uniform representation for input examples, background knowledge, and hypotheses. Given an encoding of the known background knowledge and a set of examples represented as a logical database of facts, an ILP system will derive a hypothesized logic program that entails all positive and no negative examples. Inductive programming is a related field that considers any kind of programming languages for representing hypotheses (and not only logic programming), such as functional programs.

6. Support vector machines

Support vector machines (SVMs) are a set of related supervised learning methods used for classification and regression. Given a set of training examples, each marked as belonging to one

of two categories, an SVM training algorithm builds a model that predicts whether a new example falls into one category or the other.

7. Clustering

Cluster analysis is the assignment of a set of observations into subsets (called clusters) so that observations within the same cluster are similar according to some pre designated criterion or criteria, while observations drawn from different clusters are dissimilar. Different clustering techniques make different assumptions on the structure of the data, often defined by some similarity metric and evaluated for example by internal compactness (similarity between members of the same cluster) and separation between different clusters. Other methods are based on estimated density and graph connectivity. Clustering is a method of unsupervised learning, and a common technique for statistical data analysis.

8. Bayesian networks

A Bayesian network, belief network or directed acyclic graphical model is a probabilistic graphical model that represents a set of random variables and their conditional independencies via a directed acyclic graph (DAG). For example, a Bayesian network could represent the probabilistic relationships between diseases and symptoms. Given symptoms, the network can be used to compute the probabilities of the presence of various diseases. Efficient algorithms exist that perform inference and learning.

9. Reinforcement learning

Reinforcement learning is concerned with how an agent ought to take actions in an environment so as to maximize some notion of long-term reward. Reinforcement learning algorithms attempt to find a policy that maps states of the world to the actions the agent ought to take in those states. Reinforcement learning differs from the supervised learning problem in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected.

10. Similarity and metric learning

In this problem, the learning machine is given pairs of examples that are considered similar and pairs of less similar objects. It then needs to learn a similarity function (or a distance metric function) that can predict if new objects are similar. It is sometimes used in Recommendation systems.

11. Genetic algorithms

A genetic algorithm (GA) is a search heuristic that mimics the process of natural selection, and uses methods such as mutation and crossover to generate new genotype in the hope of finding good solutions to a given problem. In machine learning, genetic algorithms found some uses in the 1980s and 1990s. Conversely, machine learning techniques have been used to improve the performance of genetic and evolutionary algorithms.

12. Rule-based machine learning

Rule-based machine learning is a general term for any machine learning method that identifies, learns, or evolves "rules" to store, manipulate or apply, knowledge. The defining characteristic of a rule- based machine learner is the identification and utilization of a set of relational rules that collectively represent the knowledge captured by the system. This is in contrast to other machine learners that commonly identify a singular model that can be universally applied to any instance in order to make a prediction. Rule-based machine learning approaches include learning classifier systems, association rule learning, and artificial immune systems.

13. Feature selection approach

Feature selection is the process of selecting an optimal subset of relevant features for use in model construction. It is assumed the data contains some features that are either redundant or irrelevant, and can thus be removed to reduce calculation cost without incurring much loss of information. Common optimality criteria include accuracy, similarity and information measures.

1. 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.

Find-s Algorithm :

1. Load Data set
2. Initialize h to the most specific hypothesis in H
3. For each positive training instance x
   - For each attribute constraint ai in h

   If the constraint ai in h is satisfied by x then do nothing
   else replace ai in h by the next more general constraint that is satisfied by x
4. Output hypothesis h

Source Code:

```
import random
import csv

def read_data(filename):
    with open(filename, 'r') as csvfile:
        datareader = csv.reader(csvfile, delimiter=',')
        traindata = []
        for row in datareader:
            traindata.append(row)
    return (traindata)

h=['phi','phi','phi','phi','phi','phi'
data=read_data('finds.csv')
def isConsistent(h,d):
        if len(h)!=len(d)-1:
            print('Number of attributes are not same in hypothesis.')
        return False
    else:
            matched=0
             for i in range(len(h)):
                    if ( (h[i]==d[i]) | (h[i]=='any') ):
                            matched=matched+1
                    if matched==len(h):
                                    return True
                    else:
                            return False
def makeConsistent(h,d):
            for i in range(len(h)):
                    if((h[i] == 'phi')):
                                    h[i]=d[i]
                    elif(h[i]!=d[i]):
                                    h[i]='any'
```

```
        return h
print('Begin : Hypothesis :',h)


print('==============================================')
for d in data:
    if d[len(d)-1]=='Yes':
        if ( isConsistent(h,d)):
            pass
        else:
                    h=makeConsistent(h,d)
    print ('Training data      :',d)
    print ('Updated Hypothesis   :',h)
    print()
    print('.....')
print('==============================================')
print('maximally sepcific data set End: Hypothesis :',h)
```

Output:

```
Begin : Hypothesis : ['phi', 'phi', 'phi', 'phi', 'phi', 'phi']
==========================================
Training data      : ['Cloudy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'Yes']
Updated Hypothesis   : ['Cloudy', 'Cold', 'High', 'Strong', 'Warm', 'Change']
----------------------------------
Training data      : ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes']
Updated Hypothesis   : ['any', 'any', 'any', 'Strong', 'Warm', 'any']
----------------------------------
Training data      : ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes']
Updated Hypothesis   : ['any', 'any', 'any', 'Strong', 'Warm', 'any']
----------------------------------
Training data      : ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']
Updated Hypothesis   : ['any', 'any', 'any', 'Strong', 'any', 'any']
----------------------------------
Training data      : ['Overcast', 'Cool', 'Normal', 'Strong', 'Warm', 'Same', 'Yes']
Updated Hypothesis   : ['any', 'any', 'any', 'Strong', 'any', 'any']
----------------------------------
==========================================
maximally sepcific data set End: Hypothesis : ['any', 'any', 'any', 'Strong', 'any', 'any']
```

OR

```
import csv
def loadCsv(filename):
    lines =
    csv.reader(open(filena
    me, "r")) dataset =
    list(lines)
    for i in
            range(le
            n(datase
            t)):
            dataset[i
            ] =
            dataset[i
            ]
    return dataset
attributes =
['Sky','Temp','Humidity','Wind','Water','Forecas
t'] print('Attributes =',attributes)
num_attributes = len(attributes)
```

Output:

```
Attributes = ['Sky', 'Temp', 'Humidity', 'Wind', 'Water', 'Forecast']
[['sky', 'Airtemp', 'Humidity', 'Wind', 'Water', 'Forecast', 'WaterSport'],
['Cloudy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'Yes'],
['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes'],
['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes'],
['Cloudy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'No'],
['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes'],
['Rain', 'Mild', 'High', 'Weak', 'Cool', 'Change', 'No'],
['Rain', 'Cool', 'Normal', 'Weak', 'Cool', 'Same', 'No'],
['Overcast', 'Cool', 'Normal', 'Strong',
'Warm', 'Same', 'Yes']] Intial Hypothesis
['0',
'0', '0', '0', '0', '0']
The Hypothesis are
2 = ['Cloudy', 'Cold', 'High', 'Strong', 'Warm', 'Change']
3 = ['?', '?', '?', 'Strong', 'Warm', '?']
4 = ['?', '?', '?', 'Strong', 'Warm', '?']
6 = ['?', '?', '?', 'Strong', '?', '?']
9 = ['?', '?', '?', 'Strong', '?', '?']
Final Hypothesis
['?', '?', '?', 'Strong', '?', '?']
```

2. 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.

Candidate-Elimination Algorithm:

1.Load data set
2.G <-maximally general hypotheses in H
3.S <- maximally specific hypotheses in H
4.For each training example
d=<x,c(x)> Case 1 : If d is a
positive example

Remove from G any hypothesis that is
inconsistent with d For each hypothesis s in S
that is not consistent with d

- Remove s from S.
- Add to S all minimal generalizations h of s such that
  - h consistent with d
  - Some member of G is more general than h
- Remove from S any hypothesis that is more general than another hypothesis in S

Case 2: If d is a negative example

Remove from S any hypothesis that is
inconsistent with d For each hypothesis g in G
that is not consistent with d

*Remove g from G.
*Add to G all minimal specializations h of g such that
  - o h consistent with d
  - o Some member of S is more specific than h
- Remove from G any hypothesis that is less general than another hypothesis in G

Source Code:

```
import numpy as np
```

```
import pandas as pd

data = pd.DataFrame(data=pd.read_csv('finds1.csv'))
concepts = np.array(data.iloc[:,0:-1])

target = np.array(data.iloc[:,-1])
def learn(concepts, target):
    specific_h = concepts[0].copy()
    print("initialization of specific_h and general_h")
    print(specific_h)
    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print(general_h)
    for i, h in enumerate(concepts):
        if target[i] == "Yes":
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    specific_h[x] = '?'
```

```
                general_h[x][x] = '?'
        if target[i] == "No":
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = '?'
    print(" steps of Candidate Elimination Algorithm",i+1)
    print("Specific_h ",i+1,"\n ")
    print(specific_h)
    print("general_h ", i+1, "\n ")
    print(general_h)

    indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?']]
    for i in indices:
        general_h.remove(['?', '?', '?', '?', '?', '?'])

    return specific_h, general_h
s_final, g_final = learn(concepts, target)
print("Final Specific_h:", s_final, sep="\n")
print("Final General_h:", g_final, sep="\n")
```

OUTPUT

```
initialization of specific_h and general_h
['Cloudy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
 steps of Candidate Elimination Algorithm 8
Specific_h  8

['?' '?' '?' 'Strong' '?' '?']
general_h  8

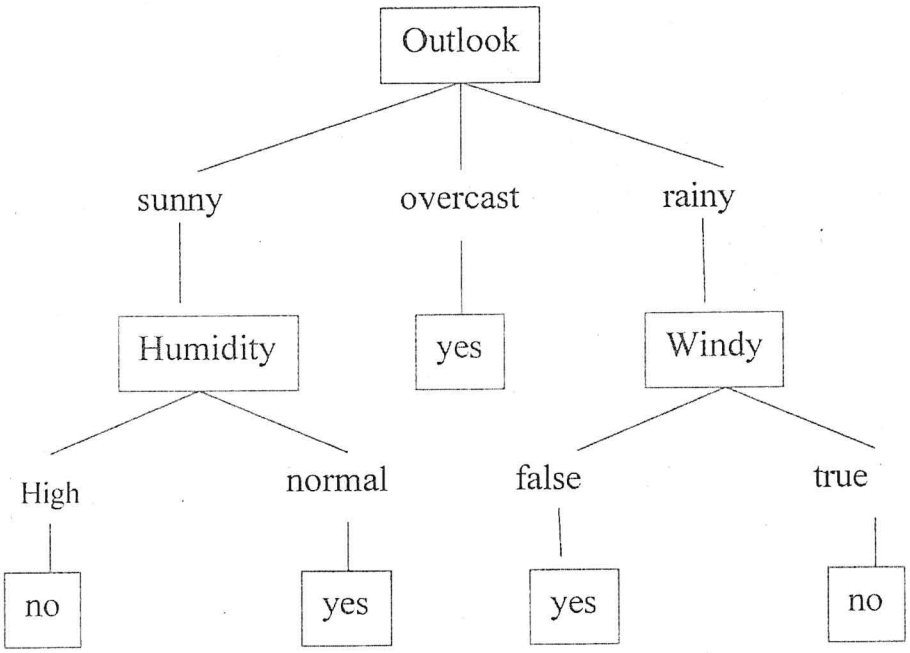
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', 'Strong', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
Final Specific_h:
['?' '?' '?' 'Strong' '?' '?']
Final General_h:
[['?', '?', '?', 'Strong', '?', '?']]
```

3. **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.**

ID3 - Algorithm

ID3*(Examples, TargetAttribute, Attributes)*

- Create a *Root* node for the tree
- If all *Examples* are positive, Return the single-node tree *Root*, with label = +
- If all *Examples* are negative, Return the single-node tree *Root*, with label = -
- If *Attributes* is empty, Return the single-node tree Root, with label = most common value of *TargetAttribute* in *Examples*
- Otherwise Begin
  - A ← the attribute from *Attributes* that best classifies *Examples*
  - The decision attribute for *Root* ←A
  - For each possible value, vi, of A,
    - Add a new tree branch below *Root*, corresponding to the test A = vi
    - Let $Examples_{vi}$ be the subset of *Examples* that have value vi for A
    - If $Examples_{vi}$ is empty
      - Then below this new branch add a leaf node with label = most common value of *TargetAttribute* in *Examples*
      - Else below this new branch add the subtree ID3($Examples_{vi}$, *TargetAttribute*, *Attributes* – {A})
- End
- Return *Root*



Source Code:

```
import numpy as np
import math
from data_loader import read_data

class Node:
    def init (self, attribute):
        self.attribute = attribute
        self.children = []
        self.answer = ""

    def __str_(self):
        return self.attribute

def subtables(data, col, delete):
    dict = {}
    items = np.unique(data[:, col])
    count = np.zeros((items.shape[0], 1), dtype=np.int32)

    for x in range(items.shape[0]):
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                count[x] += 1

    for x in range(items.shape[0]):
        dict[items[x]] = np.empty((int(count[x]), data.shape[1]), dtype="|S32")
        pos = 0
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                dict[items[x]][pos] = data[y]
                pos += 1
        if delete:
            dict[items[x]] = np.delete(dict[items[x]], col, 1)

    return items, dict

def entropy(S):
    items = np.unique(S)

    if items.size == 1:
        return 0

    counts = np.zeros((items.shape[0], 1))
    sums = 0

    for x in range(items.shape[0]):
        counts[x] = sum(S == items[x]) / (S.size * 1.0)

    for count in counts:
        sums += -1 * count * math.log(count, 2)
    return sums

def gain_ratio(data, col):
    items, dict = subtables(data, col, delete=False)

    total_size = data.shape[0]
    entropies = np.zeros((items.shape[0], 1))
    intrinsic = np.zeros((items.shape[0], 1))

    for x in range(items.shape[0]):
        ratio = dict[items[x]].shape[0]/(total_size * 1.0)
        entropies[x] = ratio * entropy(dict[items[x]][:, -1])
        intrinsic[x] = ratio * math.log(ratio, 2)
```

```
    total_entropy = entropy(data[:, -1])
    iv = -1 * sum(intrinsic)

    for x in range(entropies.shape[0]):
        total_entropy -= entropies[x]

    return total_entropy / iv

def create_node(data, metadata):
    #TODO: Co jeśli information gain jest zerowe?

    if (np.unique(data[:, -1])).shape[0] == 1:
        node = Node("")
        node.answer = np.unique(data[:, -1])[0]
        return node

    gains = np.zeros((data.shape[1] - 1, 1))

    for col in range(data.shape[1] - 1):
        gains[col] = gain_ratio(data, col)

    split = np.argmax(gains)

    node = Node(metadata[split])
    metadata = np.delete(metadata, split, 0)

    items, dict = subtables(data, split, delete=True)

    for x in range(items.shape[0]):
        child = create_node(dict[items[x]], metadata)
        node.children.append((items[x], child))

    return node

def empty(size):
    s = ""
    for x in range(size):
        s += "  "
    return s

def print_tree(node, level):
    if node.answer != "":
        print(empty(level), node.answer)
        return

    print(empty(level), node.attribute)

    for value, n in node.children:
        print(empty(level + 1), value)
        print_tree(n, level + 2)


metadata, traindata = read_data("tennis.data")

data = np.array(traindata)

node = create_node(data, metadata)

print_tree(node, 0)
```

OUTPUT:

```
outlook
  overcast
    b'yes'
  rain
```

```
    wind
      b'strong'
        b'no'
      b'weak'
        b'yes'
  sunny
    humidity
      b'high'
        b'no'
      b'normal'
        b'yes'
```

OR

```
import pandas as pd
import numpy as np
dataset= pd.read_csv('playtennis.csv',names=['outlook','temperature','humidity','wind','class',])
def entropy(target_col):
  elements,counts = np.unique(target_col,return_counts = True)
  entropy = np.sum([(-counts[i]/np.sum(counts))*np.log2(counts[i]/np.sum(counts)) for i in
range(len(elements))])
  return entropy
def InfoGain(data,split_attribute_name,target_name="class"):
   total_entropy = entropy(data[target_name])
   vals,counts= np.unique(data[split_attribute_name],return_counts=True)
   Weighted_Entropy                                                                    =
np.sum([(counts[i]/np.sum(counts))*entropy(data.where(data[split_attribute_name]==vals[i]).dr
opna()[target_name]) for i in range(len(vals))])
   Information_Gain = total_entropy - Weighted_Entropy
   return Information_Gain

def ID3(data,originaldata,features,target_attribute_name="class",parent_node_class = None):
   if len(np.unique(data[target_attribute_name])) <= 1:
      return np.unique(data[target_attribute_name])[0]
   elif len(data)==0:
      return
np.unique(originaldata[target_attribute_name])[np.argmax(np.unique(originaldata[target_attribut
e_name],return_counts=True)[1])]
   elif len(features) ==0:
      return parent_node_class
   else:
      parent_node_class                                                                =
  np.unique(data[target_attribute_name])[np.argmax(np.unique(data[target_attribute_name],return
_counts=True)[1])]
      item_values = [InfoGain(data,feature,target_attribute_name) for feature in features] #Return
the information gain values for the features in the dataset
      best_feature_index = np.argmax(item_values)
      best_feature = features[best_feature_index]
      tree = {best_feature:{}}
      features = [i for i in features if i != best_feature]
      for value in np.unique(data[best_feature]):
         value = value
         sub_data = data.where(data[best_feature] == value).dropna()
         subtree = ID3(sub_data,dataset,features,target_attribute_name,parent_node_class)
         tree[best_feature][value] = subtree
      return(tree)

tree = ID3(dataset,dataset,dataset.columns[:-1])
print(' \nDisplay Tree\n',tree)
```

OUTPUT:

```
Display Tree
 {'outlook': {'Overcast': 'Yes', 'Rain': {'wind': {'Strong': 'No', 'Weak': 'Yes'}}, 'Sunny':
{'humidity': {'High': 'No', 'Normal': 'Yes'}}}}
```

4. Build an Artificial Neural Network by implementing the Back propagation Algorithm and test the same using appropriate data sets.

function BackProp (D, η, n_{in}, n_{hidden}, n_{out})

- D is the training set consists of m pairs: $\{(x_i, y_i)^m\}$
- η is the learning rate as an example (0.1)
- n_{in}, n_{hidden} e n_{out} are the numbero of imput hidden and output unit of neural network

Make a feed-forward network with n_{in}, n_{hidden} e n_{out} units
Initialize all the weight to short randomly number (es. [-0.05 0.05])
Repeat until termination condition are verifyed:
For any sample in D:
Forward propagate the network computing the output o_u of every unit u of the network
Back propagate the errors onto the network:

- For every output unit k, compute the error δ_k: $\delta_k = o_k(1-o_k)(t_k - o_k)$
- For every hidden unit h compute the error δ_h: $\delta_h = o_h(1-o_h)\sum_{k \in outputs} w_{kh}\delta_k$
- Update the network weight w_{ji}: $w_{ji} = w_{ji} + \Delta w_{ji}$, where $\Delta w_{ji} = \eta \delta_j x_{ji}$

(x_{ji} is the input of unit j from coming from unit i)

Back propagation Algorithm:

1. Load data set
2. Assign all network inputs and output
3. Initialize all weights with small random numbers, typically between -1 and 1 repeat
   for every pattern in the
   training set Present the
   pattern to the network
// Propagated the input forward through
   the network: for each layer in the
   network
   for every node in the layer
   1.Calculate the weight sum of the inputs to the node
   2.Add the threshold to the sum
   3.Calculate the activation for
   the node end
   end

```
//   Propagate the errors backward through
      the network for every node in the
      output layer
        calculate the error
      signal end

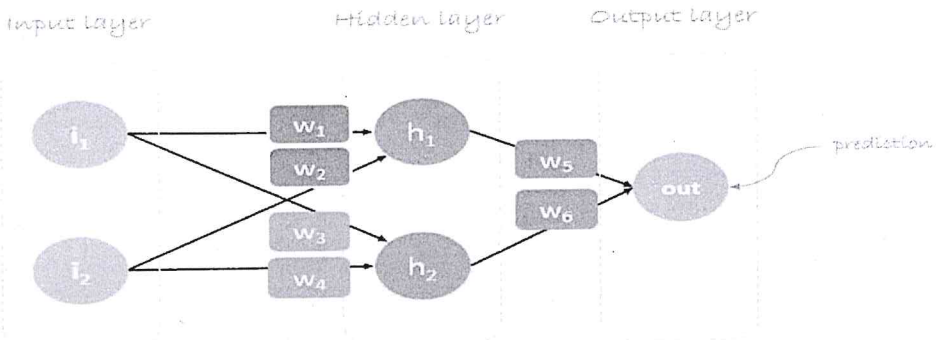
      for all hidden layers
        for every node in the layer
          1.Calculate the node's signal error
```

2.Update each node's weight in the network end
end

// Calculate Global Error
Calculate the Error
Function

end

while ((maximum number of iterations < than specified) AND (Error Function is > than specified))



- Input layer with two inputs neurons
- One hidden layer with two neurons
- Output layer with a single neuron

Source Code:

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) # maximum of X array longitudinally y = y/100

#Sigmoid Function
```

```
def sigmoid (x):
    return (1/(1 + np.exp(-x)))
#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)

                                    #Variable initialization
epoch=7000                          #Setting training iterations
lr=0.1                              #Setting learning rate
inputlayer_neurons = 2              #number of features in data set
hiddenlayer_neurons = 3              #number of hidden layers neurons
output_neurons = 1                   #number of neurons at output layer
```

```
#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))

bout=np.random.uniform(size=(1,output_neurons))
# draws a random range of numbers uniformly of dim x*y
#Forward Propagation
for i in range(epoch):
   hinp1=np.dot(X,wh)
   hinp=hinp1 + bh
   hlayer_act = sigmoid(hinp)
   outinp1=np.dot(hlayer_act,wout)
   outinp= outinp1+ bout
   output = sigmoid(outinp)
#Backpropagation
   EO = y-output
   outgrad = derivatives_sigmoid(output)
   d_output = EO* outgrad
   EH = d_output.dot(wout.T)
   hiddengrad = derivatives_sigmoid(hlayer_act)
#how much hidden layer wts contributed to error
   d_hiddenlayer = EH * hiddengrad
   wout += hlayer_act.T.dot(d_output) *lr
# dotproduct of nextlayererror and currentlayerop
   bout += np.sum(d_output, axis=0,keepdims=True) *lr
   wh += X.T.dot(d_hiddenlayer) *lr
#bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr
print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

Output:

```
Input:
[[ 0.66666667 1.        ]
 [ 0.33333333 0.55555556]
 [ 1.         0.66666667]]

Actual Output:
[[ 0.92]
 [ 0.86]
 [ 0.89]]
Predicted Output:
 [[ 0.89559591]
 [ 0.88142069]
 [ 0.8928407 ]]
```

5. Implement k-nearest neighbours classification using python

ALGORITHM:

Step 1: Load the data
Step 2: Initialize the value of k
Step 3: For getting the predicted class, iterate from 1 to total number of training data points

i) Calculate the distance between test data and each row of training data. Here we will use Euclidean distance as our distance metric since it's the most popular method. The other metrics that can be used are Chebyshev, cosine, etc.
ii) Sort the calculated distances in ascending order based on distance values 3. Get top k rows from the sorted array
iii) Get the most frequent class of these rows i.e. Get the labels of the selected K entries
iv) Return the predicted class If regression, return the mean of the K labels If classification, return the mode of the K labels

- If regression, return the mean of the K labels
- If classification, return the mode of the K labels

Step 4: End.

PROGRAM

```
import numpy as np
from sklearn import datasets

iris = datasets.load_iris()
data = iris.data
labels = iris.target

for i in [0, 79, 99, 101]:
        print(f"index: {i:3}, features: {data[i]}, label: {labels[i]}")

np.random.seed(42)
indices = np.random.permutation(len(data))
 n_training_samples = 12
learn_data = data[indices[:-n_training_samples]]
learn_labels = labels[indices[:-n_training_samples]]
```

```
colours = ("r", "g", "y")
 fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for iclass in range(3):
        ax.scatter(X[iclass][0], X[iclass][1], X[iclass][2], c=colours[iclass])
plt.show()
#-----------------------------------------------------------------

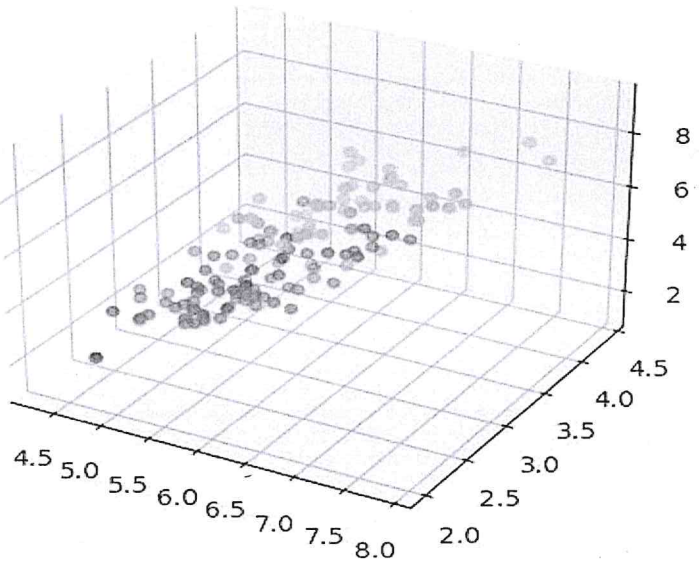
def distance(instance1, instance2):
    """ Calculates the Eucledian distance between two instances"""
        return np.linalg.norm(np.subtract(instance1, instance2))

def get_neighbors(training_set, labels, test_instance, k, distance):
    """
    get_neighors calculates a list of the k nearest neighbors  of an instance 'test_instance'.
    The function returns a list of k 3-tuples. Each 3-tuples consists of (index, dist, label)
    """
    distances = []
    for index in range(len(training_set)):
        dist = distance(test_instance, training_set[index])
        distances.append((training_set[index], dist, labels[index]))
    distances.sort(key=lambda x: x[1])
    neighbors = distances[:k]
    return neighbors
```

```
for i in range(5):
    neighbors = get_neighbors(learn_data, learn_labels, test_data[i], 3, distance=distance)
print("Index:        ",i,'\n',
      "Testset Data: ",test_data[i],'\n',
      "Testset Label: ",test_labels[i],'\n',
      "Neighbors:    ",neighbors,'\n')
```

OUTPUT:

```
(base) dohathi@dohathi-Compaq-15-Notebook-PC:~/ML_LAB$ python KNN.py
index:   0, features: [5.1 3.5 1.4 0.2], label: 0
index:  79, features: [5.7 2.6 3.5 1. ], label: 1
index:  99, features: [5.7 2.8 4.1 1.3], label: 1
index: 101, features: [5.8 2.7 5.1 1.9], label: 2
The first samples of our learn set:
index  data                label
   0   [6.1 2.8 4.7 1.2]     1
   1   [5.7 3.8 1.7 0.3]     0
   2   [7.7 2.6 6.9 2.3]     2
   3   [6.  2.9 4.5 1.5]     1
   4   [6.8 2.8 4.8 1.4]     1
The first samples of our test set:
index  data                label
   0   [6.1 2.8 4.7 1.2]     1
   1   [5.7 3.8 1.7 0.3]     0
   2   [7.7 2.6 6.9 2.3]     2
   3   [6.  2.9 4.5 1.5]     1
   4   [6.8 2.8 4.8 1.4]     1
```



```
Index:          2
 Testset Data:   [6.3 2.3 4.4 1.3]
 Testset Label:  1
 Neighbors:       [(array([6.2, 2.2, 4.5, 1.5]), 0.26457513110645864, 1),
 (array([6.3, 2.5, 4.9, 1.5]), 0.574456264653803, 1), (array([6. , 2.2, 4
. , 1. ]), 0.5916079783099617, 1)]

Index:          3
 Testset Data:   [6.4 2.9 4.3 1.3]
 Testset Label:  1
 Neighbors:       [(array([6.2, 2.9, 4.3, 1.3]), 0.20000000000000018, 1),
 (array([6.6, 3. , 4.4, 1.4]), 0.2645751311064587, 1), (array([6.6, 2.9,
4.6, 1.3]), 0.3605551275463984, 1)]

Index:          4
 Testset Data:   [5.6 2.8 4.9 2. ]
 Testset Label:  2
 Neighbors:       [(array([5.8, 2.7, 5.1, 1.9]), 0.31622776601683755, 2),
 (array([5.8, 2.7, 5.1, 1.9]), 0.31622776601683755, 2), (array([5.7, 2.5,
 5. , 2. ]), 0.33166247903553986, 2)]
```

6. Implementation of Naïve Bayes classifier algorithm

Implementation of Naïve Bayes classifier algorithm

```
import math
import random
import csv
def encode_class(mydata):
classes = []
for i in range(len(mydata)):
    if mydata[i][-1] not in classes:
    classes.append(mydata[i][-1])
  for i in  range(len(classes)):
  for j in range(len(mydata)):
      if mydata[j][-1] == classes[i]:
      mydata[j][-1] = i
  return mydata
def splitting(mydata, ratio):
  train_num = int(len(mydata) * ratio)
  train = []
  test = list(mydata)
  while len(train) < train_num:
    index = random.randrange(len(test))
    train.append(test.pop(index))
  return train, test
def groupUnderClass(mydata):
  dict = {}
   for i in range(len(mydata)):
     if (mydata[i][-1] not in dict):
     dict[mydata[i][-1]] = []
     dict[mydata[i][-1]].append(mydata[i])
     return dic
```

```
    return sum(numbers) / float(len(numbers))

def std_dev(numbers):
avg = mean(numbers)
variance = sum([pow(x - avg, 2) for x in numbers]) / float(len(numbers) - 1)return
math.sqrt(variance)


def MeanAndStdDev(mydata):
info = [(mean(attribute), std_dev(attribute)) for attribute in zip(*mydata)]

del info[-1]
return info


def MeanAndStdDevForClass(mydata):
info = {}
dict = groupUnderClass(mydata)
for classValue, instances in dict.items():

      info[classValue] = MeanAndStdDev(instances)
return info


def calculateGaussianProbability(x, mean, stdev):
    expo = math.exp(-(math.pow(x - mean, 2) / (2 * math.pow(stdev, 2))))

    return (1 / (math.sqrt(2 * math.pi) * stdev)) * expo
def calculateClassProbabilities(info, test):
    probabilities = {}

for classValue, classSummaries in info.items():
    probabilities[classValue] = 1
for i in range(len(classSummaries)):

      mean, std_dev = classSummaries[i]x = test[i]
probabilities[classValue] *= calculateGaussianProbability(x, mean, std_dev)
```

7. Implement the linear regression algorithm

AIM: - Implement the linear regression algorithm.

SOURCE CODE: -

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Sample data
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 5, 7, 8])

# Create and fit the model
model = LinearRegression()
model.fit(X, y)

# Predict the output
X_test = np.array([[6], [7], [8]])
y_pred = model.predict(X_test)

# Print the coefficients and intercept
print("Coefficients:", model.coef_)
print("Intercept:", model.intercept_)

# Print the predictions
print("Predictions:", y_pred)
```

OUTPUT:

Coefficients: [1.2]

Intercept: 0.6

Predictions: [8.4 9.6 10.8]

8. Implement the logistic regression algorithm

AIM: - Implement the logistic regression algorithm.

SOURCE CODE: -

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Sample data
X = np.array([[1, 2], [2, 3], [3, 1], [4, 3], [5, 3]])
y = np.array([0, 0, 0, 1, 1])

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create and fit the model
model = LogisticRegression()
model.fit(X_train, y_train)
# Predict the output
y_pred = model.predict(X_test)
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

OUTPUT:

Accuracy: 0.5

9.Implementation of Linear & Polynomial Regression

Linear Regression

```
Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
   random_state=0)

# Initialize and train the Linear Regression model
linear_regressor = LinearRegression()
linear_regressor.fit(X_train, y_train)

# Make predictions
y_pred_linear = linear_regressor.predict(X_test)

# Plot the results
plt.scatter(X_test, y_test, color='blue', label='Actual data')
plt.plot(X_test, y_pred_linear, color='red', label='Linear regression
   line')
plt.title('Linear Regression')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()

# Evaluate the model
mse_linear = mean_squared_error(y_test, y_pred_linear)
print(f"Mean Squared Error (Linear Regression): {mse_linear}")
```

Polynomial Regression

For **Polynomial Regression**, we need to transform the input data to include polynomial terms.

```
Python

# Transform the feature data to include polynomial terms (degree 2 in this
   case)
polynomial_features = PolynomialFeatures(degree=2)
X_poly = polynomial_features.fit_transform(X)

# Split the data again for polynomial regression
X_train_poly, X_test_poly, y_train_poly, y_test_poly =
   train_test_split(X_poly, y, test_size=0.2, random_state=0)

# Initialize and train the Polynomial Regression model
poly_regressor = LinearRegression()
poly_regressor.fit(X_train_poly, y_train_poly)

# Make predictions
y_pred_poly = poly_regressor.predict(X_test_poly)

# Plot the results
plt.scatter(X_test, y_test, color='blue', label='Actual data')
plt.scatter(X_test, y_pred_poly, color='green', label='Polynomial
   regression')
plt.title('Polynomial Regression')
plt.xlabel('X')
```

```
plt.ylabel('y')
plt.legend()
plt.show()

# Evaluate the model
mse_poly = mean_squared_error(y_test_poly, y_pred_poly)
print(f"Mean Squared Error (Polynomial Regression): {mse_poly}")
```

10.Python Program to Implement Estimation & MAximization Algorithm

Now, let's implement the EM algorithm, iterating through the E-step and M-step:

```
# EM Algorithm

def e_step(data, mean1, mean2, cov1, cov2, pi1, pi2):

  # Calculate the responsibilities (E-step)

  rv1 = multivariate_normal(mean1, cov1)

  rv2 = multivariate_normal(mean2, cov2)

  gamma1 = pi1 * rv1.pdf(data)

  gamma2 = pi2 * rv2.pdf(data)

  gamma_sum = gamma1 + gamma2

  gamma1 /= gamma_sum

  gamma2 /= gamma_sum

  return gamma1, gamma2

def m_step(data, gamma1, gamma2):

  # Update the parameters (M-step)

  N1 = np.sum(gamma1)

  N2 = np.sum(gamma2)

  mean1_new = np.sum(gamma1[:, np.newaxis] * data, axis=0) / N1

  mean2_new = np.sum(gamma2[:, np.newaxis] * data, axis=0) / N2


  cov1_new = (gamma1[:, np.newaxis] * (data - mean1_new)).T @ (data - mean1_new) / N1

  cov2_new = (gamma2[:, np.newaxis] * (data - mean2_new)).T @ (data - mean2_new) / N2
```

```
    pi1_new = N1 / len(data)

    pi2_new = N2 / len(data)


    return mean1_new, mean2_new, cov1_new, cov2_new, pi1_new, pi2_new

# Iterate until convergence

for _ in range(100):  # Limit iterations to prevent infinite loop

    gamma1, gamma2 = e_step(data, mean1_guess, mean2_guess, cov1_guess, cov2_guess, pi1, pi2)

    mean1_guess, mean2_guess, cov1_guess, cov2_guess, pi1, pi2 = m_step(data, gamma1, gamma2)
```

- **E-step**: The algorithm calculates the "responsibilities" for each data point, indicating the probability of the point belonging to each Gaussian component.
- **M-step**: The parameters (means, covariances, and weights) are updated based on these probabilities.

Plot the Final Estimated Density

Finally, we visualize the estimated Gaussian components based on the final parameter values:

```
# Visualize the estimated density

x, y = np.meshgrid(np.linspace(0, 15, 100), np.linspace(0, 15, 100))

pos = np.dstack((x, y))

rv1 = multivariate_normal(mean1_guess, cov1_guess)

rv2 = multivariate_normal(mean2_guess, cov2_guess)

plt.contour(x, y, rv1.pdf(pos), colors='blue', alpha=0.5)

plt.contour(x, y, rv2.pdf(pos), colors='red', alpha=0.5)

plt.scatter(data[:, 0], data[:, 1], s=5)

plt.title('Final Estimated Gaussian Components')

plt.show()
```

11.Credit Score Classification Program Using Python

In this section, I will guide you through creating a program for credit score classification using Python and machine learning. The process involves several steps, including data preparation, exploratory data analysis, model training, and evaluation.

Step 1: Import Necessary Libraries

First, we need to import the required libraries for data manipulation, visualization, and machine learning.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix
```

Step 2: Load the Dataset

Next, load the dataset that contains the credit score information. Ensure that the dataset is in CSV format and accessible.

```
# Load the dataset
data = pd.read_csv("train.csv")
print(data.head())
```

Step 3: Data Preprocessing

Before training a model, we need to preprocess the data. This includes checking for null values and encoding categorical variables.

```
# Check for null values
print(data.isnull().sum())

# Drop any rows with null values (if necessary)
data.dropna(inplace=True)

# Encode categorical variables using Label Encoding
label_encoder = LabelEncoder()
data['Occupation'] = label_encoder.fit_transform(data['Occupation'])
data['Credit_Mix'] = label_encoder.fit_transform(data['Credit_Mix'])
data['Payment_of_Min_Amount'] =
label_encoder.fit_transform(data['Payment_of_Min_Amount'])
data['Payment_Behaviour'] =
label_encoder.fit_transform(data['Payment_Behaviour'])
data['Credit_Score'] = label_encoder.fit_transform(data['Credit_Score'])  #
Target variable

# Display processed data info
print(data.info())
```

Step 4: Feature Selection

Select relevant features that will be used to predict the credit score.

```
# Define features and target variable
X = data.drop(['ID', 'Customer_ID', 'Credit_Score'], axis=1)  # Features
(excluding ID columns and target)
y = data['Credit_Score']  # Target variable (encoded)
```

Step 5: Split the Data into Training and Testing Sets

We will split our dataset into training and testing sets to evaluate our model's performance.

```
# Split the dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Step 6: Train a Machine Learning Model

We will use a Random Forest Classifier for this task due to its effectiveness in classification problems.

```
# Initialize the Random Forest Classifier
model = RandomForestClassifier(n_estimators=100)

# Fit the model on training data
model.fit(X_train, y_train)
```

Step 7: Make Predictions

After training the model, we can make predictions on the test set.

```
# Make predictions on test set
y_pred = model.predict(X_test)
```

Step 8: Evaluate Model Performance

Finally, we will evaluate our model's performance using confusion matrix and classification report.

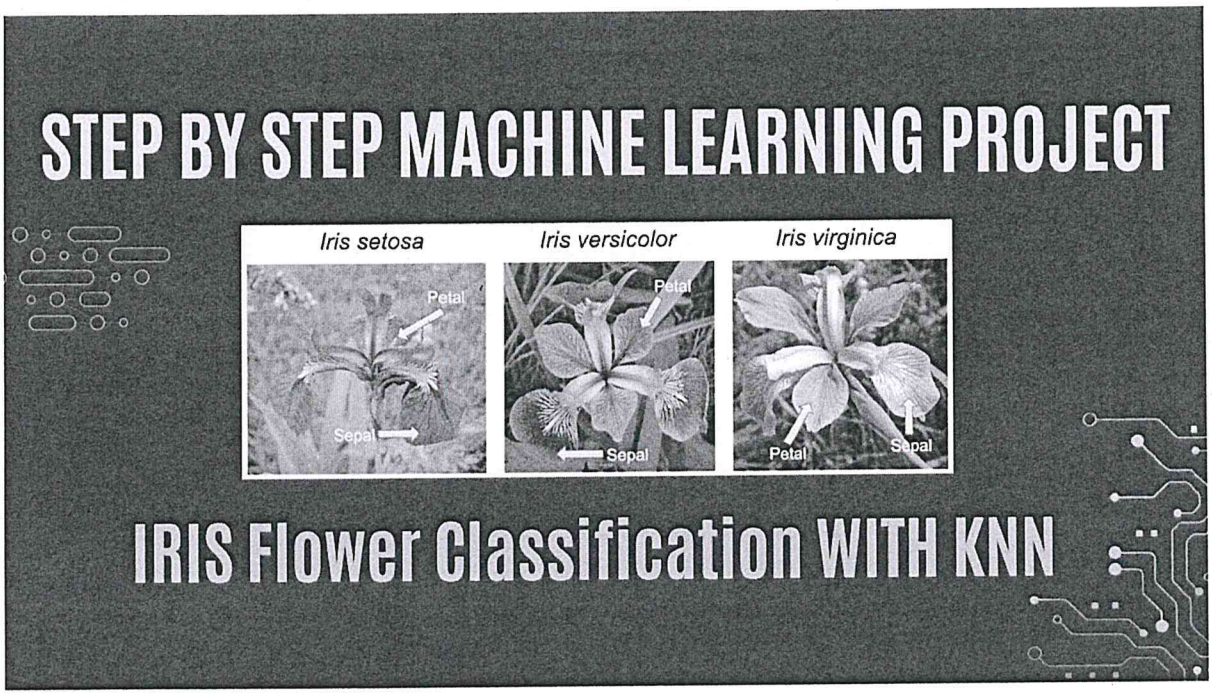
```
# Generate confusion matrix and classification report
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print("Confusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", class_report)

# Visualize confusion matrix using Seaborn heatmap for better understanding
plt.figure(figsize=(10,7))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

This program provides a comprehensive approach to credit score classification using machine learning techniques in Python. It covers all essential steps from loading data to evaluating model performance.

12. Iris Flower Classification using KNN



Iris Flower Classification using KNN

Introduction to the Iris Dataset and KNN Algorithm

The Iris dataset is a well-known dataset in the field of machine learning and statistics, commonly used for classification tasks. It consists of 150 samples of iris flowers, with each sample characterized by four features: sepal length, sepal width, petal length, and petal width. The dataset includes three species of iris flowers: Iris setosa, Iris versicolor, and Iris virginica, with 50 samples from each species.

The K-Nearest Neighbors (KNN) algorithm is a simple yet effective classification technique that classifies data points based on the classes of their nearest neighbors in the feature space. The algorithm operates under the principle that similar data points are likely to belong to the same class.

Step-by-Step Process for Classifying Iris Flowers Using KNN

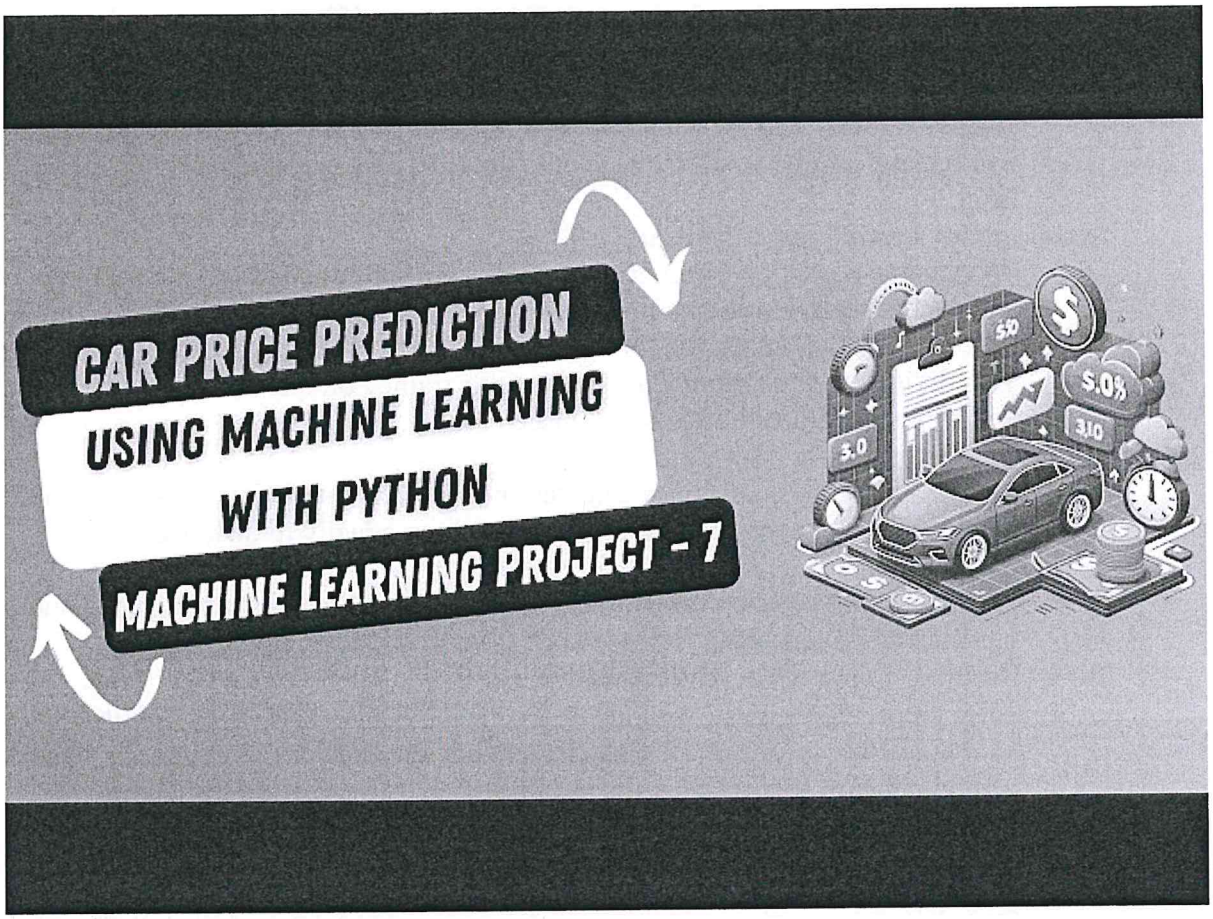
1. **Data Collection and Preparation**
   - The first step involves loading the Iris dataset into a DataFrame using libraries such as Pandas in Python. This can be done using `pd.read_csv()` if the data is stored in a CSV file.
   - After loading the data, it is essential to explore it to understand its structure and contents. Functions like `describe()` provide statistical summaries while `value_counts()` can show counts of unique values for categorical variables.
2. **Data Visualization**
   - Visualization plays a crucial role in understanding patterns within the data. Libraries like Matplotlib can be used to create histograms and scatter plots.

- Histograms can illustrate the distribution of individual features (e.g., sepal length), while scatter plots can help visualize relationships between pairs of features (e.g., sepal length vs. sepal width) across different species.

3. **Dividing Data into Input and Output Values**
   - In this step, we separate the features (input values) from the target variable (output values). The first four columns represent features (sepal length, sepal width, petal length, petal width), while the fifth column represents the species label.
4. **Training and Testing Data Division**
   - To evaluate model performance accurately, we split the dataset into training and testing sets. A common practice is to use a 70:30 ratio where 70% of the data is used for training and 30% for testing.
   - This division helps ensure that our model is trained on one subset of data while being evaluated on another unseen subset.
5. **Feature Scaling**
   - Since KNN relies on distance calculations between data points, it is important to normalize or scale features so they contribute equally to distance computations.
   - Min-max normalization transforms feature values into a range between 0 and 1 using techniques like `MinMaxScaler` from Scikit-learn.
6. **Model Building Using KNN Algorithm**
   - After preparing the data, we implement the KNN classifier using Scikit-learn's `KNeighborsClassifier`. We specify parameters such as `n_neighbors`, which determines how many neighbors will influence classification decisions.
7. **Model Training**
   - The model is trained using the training dataset by calling `.fit(X_train, y_train)` where `X_train` contains input features and `y_train` contains corresponding labels.
8. **Making Predictions**
   - Once trained, predictions can be made on both test data (`y_pred = knn.predict(X_test)`) and new instances by providing new feature arrays.
9. **Model Evaluation**
   - To assess model performance, metrics such as accuracy score and confusion matrix are calculated.
     - **Accuracy Score** measures how many predictions were correct out of total predictions made.
     - A **Confusion Matrix** provides insights into true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).
     - Additionally, a classification report can summarize precision, recall, and F1-score for each class.
10. **Conclusion**
    - By following these steps systematically—from loading data through evaluation—we can effectively classify iris flowers based on their measurements using KNN.

Final Answer: The process described above outlines how to classify iris flowers using K-Nearest Neighbors (KNN) algorithm effectively utilizing Python libraries such as NumPy, Pandas, Matplotlib, and Scikit-learn.

13. Car Price Prediction Model using Python



Car Price Prediction Model using Python

To build a car price prediction model using Python, we will follow a systematic approach that involves data collection, preprocessing, exploratory data analysis (EDA), model training, and evaluation. Below is a detailed step-by-step guide to creating this model.

Step 1: Data Collection

The first step in building the car price prediction model is to gather the dataset. For this project, we will use a dataset available on Kaggle that contains various features of used cars such as their prices, specifications, and other relevant attributes. The dataset can be accessed at Kaggle Vehicle Dataset.

Step 2: Data Preprocessing

Once the dataset is collected, it needs to be preprocessed to ensure it is clean and suitable for analysis. This includes:

- **Loading the Data:** Use pandas to load the CSV file into a DataFrame.
- `import pandas as pd`
-

```
df = pd.read_csv('path_to_your_file.csv')
```

- **Handling Missing Values:** Check for any missing values in the dataset and decide how to handle them (e.g., removing or imputing).
- ```
  df.isnull().sum()  # Check for missing values
  df.dropna(inplace=True)  # Remove rows with missing values
  ```

- **Data Type Conversion:** Ensure that all columns have the correct data types. For instance, if the price column is read as an object type due to non-numeric characters, convert it to numeric.

  ```
  df['price'] = pd.to_numeric(df['price'], errors='coerce')
  ```

- **Feature Encoding:** Convert categorical variables into numerical format using techniques like one-hot encoding.

  ```
  df = pd.get_dummies(df, columns=['Fuel_Type', 'Seller_Type',
  'Transmission'], drop_first=True)
  ```

Step 3: Exploratory Data Analysis (EDA)

Perform EDA to understand patterns in the data:

- **Visualizations:** Use libraries like Matplotlib and Seaborn to create visual representations of relationships between features and prices.

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.scatterplot(x='Kms_Driven', y='price', data=df)
plt.title('Price vs Kms Driven')
plt.show()
```

- **Correlation Analysis:** Analyze correlations between different features and the target variable (price).

```
correlation_matrix = df.corr()
sns.heatmap(correlation_matrix, annot=True)
plt.show()
```

Step 4: Model Training

After preprocessing and exploring the data, we can proceed with training a machine learning model. A common choice for regression tasks like this is Linear Regression.

- **Splitting the Dataset:** Divide your dataset into training and testing sets.

```
from sklearn.model_selection import train_test_split

X = df.drop('price', axis=1)    # Features
y = df['price']                  # Target variable

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```
```

- **Training the Model:**

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)
```

Step 5: Model Evaluation

Evaluate your model's performance using metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), or R-squared score.

```
from sklearn.metrics import mean_absolute_error, mean_squared_error

y_pred = model.predict(X_test)

mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)

print(f'MAE: {mae}, MSE: {mse}')
```

Step 6: Deployment

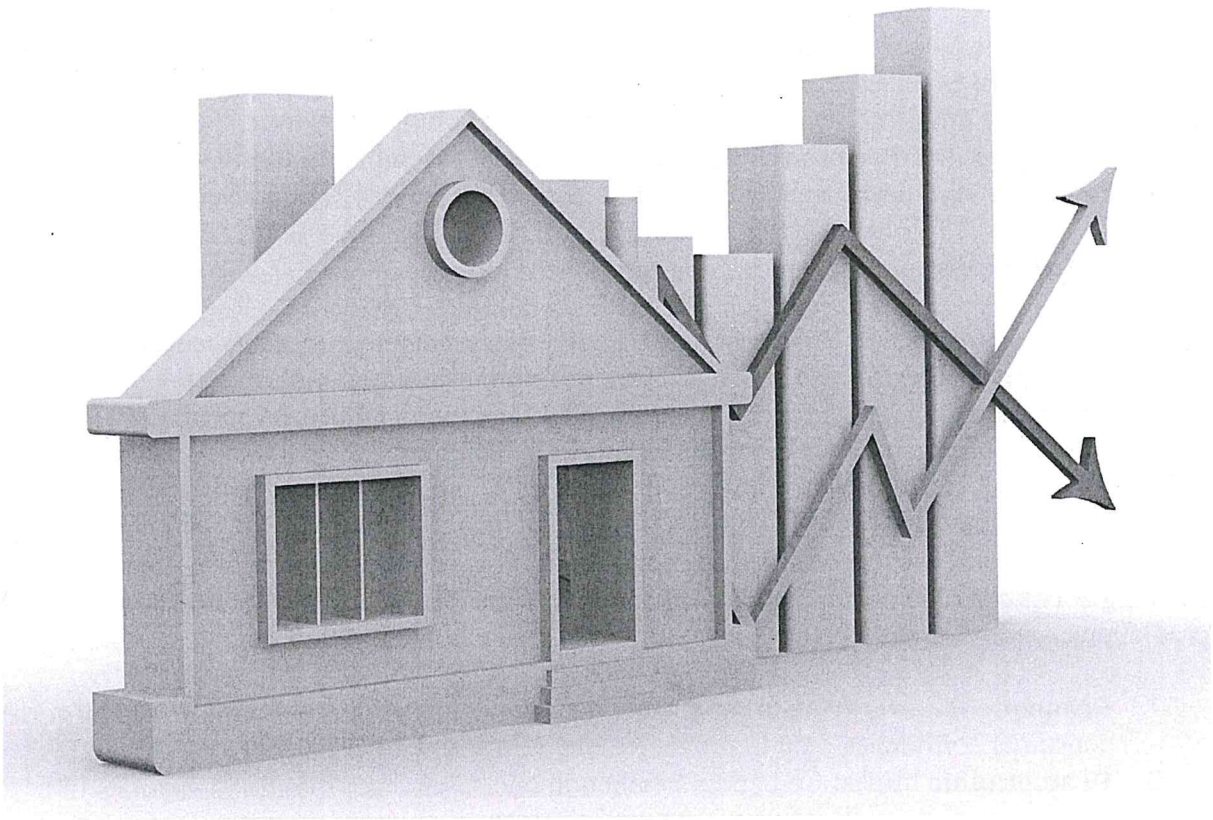
Once satisfied with your model's performance:

- Create a Flask web application that allows users to input car features and receive predicted prices.
- Deploy your application on platforms like Heroku or AWS for public access.

You can view an example of a deployed application at Heroku Car Price Prediction App.

In summary, building a car price prediction model involves collecting relevant data from sources like Kaggle, preprocessing it effectively by handling missing values and encoding categorical variables, performing exploratory data analysis to gain insights into feature relationships with prices, training a regression model such as Linear Regression on processed data while evaluating its performance through appropriate metrics before deploying it via web applications.

14.House price Prediction



House Price Prediction

The housing market has experienced significant fluctuations over the past few years, influenced by various economic factors such as mortgage rates, inflation, and supply-demand dynamics. As of December 2024, predictions regarding house prices can be analyzed through several key trends and data points.

Current Market Conditions

As of June 2024, the median existing-home sale price in the United States reached approximately $426,900, marking it as the highest recorded price by the National Association of Realtors (NAR). For new construction homes, the median sale price was slightly lower at around $417,300. These figures indicate that home prices have remained resilient despite rising mortgage rates and a slowdown in sales activity.

Supply and Demand Dynamics

The inventory of homes for sale is crucial in understanding future price movements. Currently, while there is an increase in housing supply, it remains insufficient to meet demand. The NAR reported a 4.1-month supply of unsold existing homes as of June 2024; a balanced market typically requires a 5- to 6-month supply. This persistent imbalance suggests

that home prices are likely to hold steady or even appreciate in certain markets where demand outstrips available inventory.

Impact of Mortgage Rates

Mortgage rates have been a significant factor influencing home prices. As of late 2024, mortgage rates are expected to range between 6% and 7%. High mortgage rates have constrained buyer affordability and reduced overall sales volume. However, if these rates begin to decline gradually—projected to potentially reach around 5.5% to 6% within two years—this could stimulate demand and lead to upward pressure on home prices.

Regional Variations

It's important to note that housing markets are not uniform across the country; local fundamentals play a critical role in determining price trends. Areas with strong job growth, population increases, or limited housing supply may see more significant appreciation compared to regions facing economic challenges or oversupply.

Long-Term Outlook

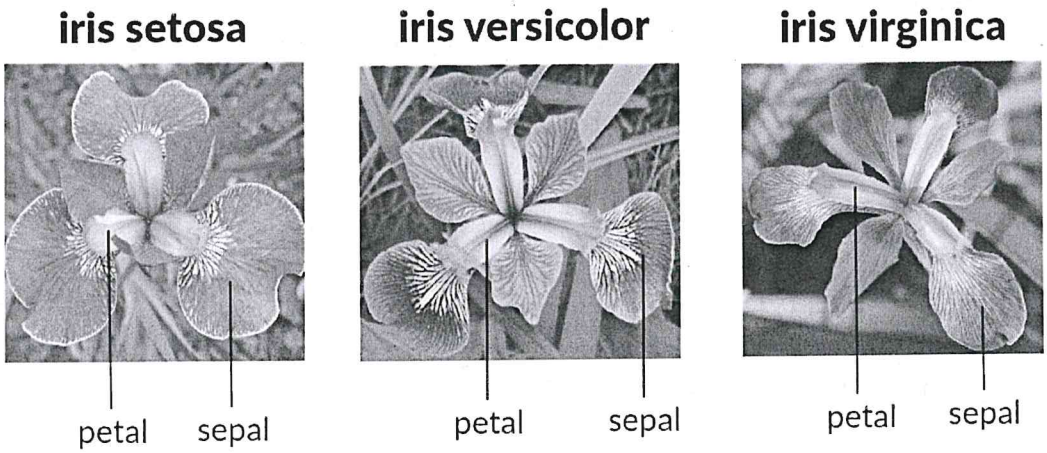
Looking ahead over the next five years until 2029, several factors will shape house price predictions:

1. **Economic Recovery:** As inflation stabilizes and interest rates potentially decrease, consumer confidence may improve, leading to increased purchasing activity.
2. **Demographic Shifts:** Changing demographics—including millennials entering the housing market—will influence demand patterns.
3. **Climate Change Considerations:** The rising costs associated with climate change may also impact housing desirability and pricing in vulnerable areas.

Overall, while short-term fluctuations may occur due to economic conditions and interest rate changes, long-term projections suggest that house prices will likely remain stable or appreciate modestly due to ongoing supply constraints and demographic shifts favoring homeownership.

In summary, **house prices are expected to hold their value or appreciate slightly over the next few years**, particularly if mortgage rates decline and inventory remains limited.

15.NAÏVE IRIS Classification



NAÏVE IRIS CLASSIFICATION

Introduction to Naïve Bayes Classifier for Iris Classification

The classification of Iris flowers using the Naïve Bayes algorithm is a well-known example in machine learning. The dataset consists of measurements from three species of Iris flowers: Iris setosa, Iris versicolor, and Iris virginica. Each flower is characterized by four features: sepal length, sepal width, petal length, and petal width. The goal is to classify these flowers based on their features using the Naïve Bayes classifier.

Understanding the Dataset

The Iris dataset contains 150 instances with equal representation from each species (50 samples per species). The four features are continuous numerical values measured in centimeters. For effective classification using Naïve Bayes, these continuous values can be converted into categorical values (e.g., "short," "medium," "long") or used directly as continuous data depending on the implementation.

Steps in the Naïve Bayes Classification Process

1. **Data Preparation**:
   - Load the dataset and split it into training and testing sets. Typically, 80% of the data is used for training while 20% is reserved for testing.
   - This can be done using libraries such as `scikit-learn`, which provides functions like `train_test_split`.
2. **Model Initialization**:
   - Initialize a Gaussian Naive Bayes classifier (`GaussianNB`), which assumes that the features follow a Gaussian distribution.
3. **Training the Model**:
   - Fit the model to the training data using the `fit` method. During this phase, the model calculates prior probabilities for each class and likelihoods for each feature given each class.

4. **Making Predictions**:
    - Use the trained model to predict classes for the test data with the `predict` method.
5. **Evaluating Model Performance**:
    - Assess how well the model performs by comparing predicted classes against actual classes in the test set using metrics such as accuracy score.

Implementation Example

Here's a step-by-step breakdown of how this process can be implemented in Python:

```
# Import necessary modules
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import GaussianNB

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features: sepal length, sepal width, petal length, petal
width
y = iris.target  # Target: species of iris

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Initialize a Gaussian Naive Bayes classifier
clf = GaussianNB()

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)

# Print accuracy score of predictions
print("Accuracy:", accuracy_score(y_test, y_pred))
```

In this code:

- We first load and prepare our dataset.
- We then split it into training and testing subsets.
- A Gaussian Naive Bayes classifier is initialized and trained on our training data.
- Finally, we make predictions on our test set and evaluate performance through accuracy.

Conclusion

The Naïve Bayes classifier effectively categorizes iris flowers based on their physical characteristics by leveraging probabilistic principles underpinned by Bayes' theorem. Its simplicity and efficiency make it an excellent choice for classification tasks like this one.

16. Comparison of Classification Algorithms

Technique	Primary Problem	Predictors	Power	Raw Implementation	Interpretability	Regression Also	Normalization
k-NN	multiclass or binary	numeric	medium	easy	good	no	required
perceptron	binary	numeric	low	easy	good	no	no
logistic regression	binary	numeric	low	easy	good	no	no
linear discriminant analysis	binary	numeric	low	medium	medium	no	no
naive Bayes	multiclass or binary	categorical	medium	medium	good	no	required
decision tree	multiclass or binary	numeric or categorical	high	difficult	good	yes	no
random forest	multiclass or binary	numeric or categorical	high	difficult	good	yes	no
adaboost	multiclass or binary	numeric or categorical	high	medium	medium	yes	usually
support vector machine	binary	numeric or categorical	high	very difficult	medium	no	yes
neural network	multiclass or binary	numeric or categorical	very high	very difficult	weak	yes	yes

Comparison of Classification Algorithms

When comparing classification algorithms in machine learning, it is essential to consider several factors that influence their performance and suitability for specific tasks. These factors include the nature of the data, the complexity of the model, interpretability, training time, and accuracy. Below is a detailed comparison of some of the most commonly used classification algorithms: Logistic Regression, Decision Trees, Random Forests, Support Vector Machines (SVM), Naive Bayes, and K-Nearest Neighbors (KNN).

1. Logistic Regression

Logistic regression is a statistical method used for binary classification problems. It predicts the probability that an instance belongs to a particular class by applying a logistic function to a linear combination of input features.

- **Strengths**:
  - Simple to implement and interpret.
  - Works well with linearly separable data.
  - Provides probabilities for predictions.
- **Weaknesses**:
  - Assumes linearity between independent variables and the log odds of the dependent variable.
  - Not suitable for complex relationships or multi-class problems without modifications.

2. Decision Trees

Decision trees are a non-parametric supervised learning method used for classification and regression tasks. They split the data into subsets based on feature values, creating a tree-like model of decisions.

- **Strengths**:
  - Easy to understand and visualize.
  - Handles both numerical and categorical data.
  - Requires little data preprocessing (e.g., no need for normalization).
- **Weaknesses**:
  - Prone to overfitting if not pruned properly.
  - Can be unstable; small changes in data can lead to different tree structures.

3. Random Forest

Random forests are an ensemble learning method that constructs multiple decision trees during training and outputs the mode of their predictions for classification tasks.

- **Strengths**:
  - Reduces overfitting compared to individual decision trees.
  - Handles large datasets with higher dimensionality well.
  - Provides feature importance scores.
- **Weaknesses**:
  - More complex than single decision trees; harder to interpret.
  - Requires more computational resources due to multiple trees.

4. Support Vector Machines (SVM)

Support vector machines are supervised learning models that analyze data for classification and regression analysis. SVM works by finding the hyperplane that best separates different classes in high-dimensional space.

- **Strengths**:
  - Effective in high-dimensional spaces.
  - Works well with clear margin separation between classes.
- **Weaknesses**:
  - Less effective on very large datasets due to high training time.
  - Requires careful tuning of parameters like kernel choice and regularization.

5. Naive Bayes

Naive Bayes classifiers are based on applying Bayes' theorem with strong (naive) independence assumptions between features. They are particularly suited for large datasets.

- **Strengths**:
  - Fast training and prediction times.
  - Performs well with high-dimensional data such as text classification.
- **Weaknesses**:

- The assumption of feature independence rarely holds true in real-world scenarios, which can affect performance.

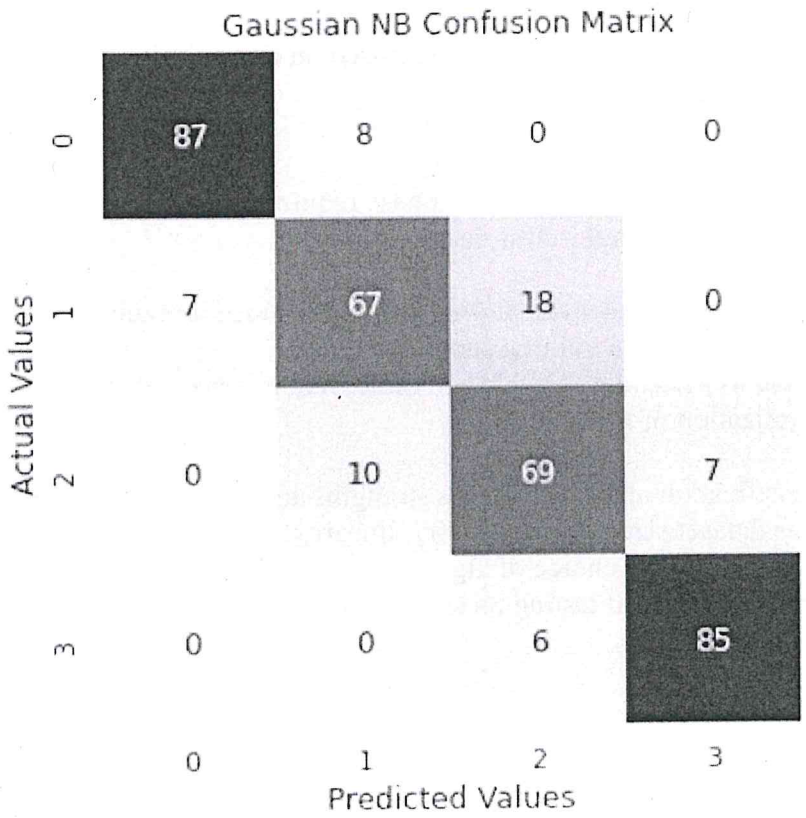
6. K-Nearest Neighbors (KNN)

KNN is a simple algorithm that classifies instances based on the majority class among its k-nearest neighbors in the feature space.

- **Strengths**:
  - Simple to implement; no training phase required (instance-based).
  - Naturally handles multi-class cases.
- **Weaknesses**:
  - Computationally expensive during prediction since it requires distance calculations from all training samples.
  - Sensitive to irrelevant features and scale; requires normalization or standardization of features.

In summary, each classification algorithm has its strengths and weaknesses depending on various factors such as dataset size, dimensionality, linearity, interpretability requirements, and computational efficiency. The choice of algorithm should be guided by these considerations along with empirical testing on specific datasets to determine which performs best in practice.

17. Mobile Price Classification using Python



Mobile Price Classification using Python

Introduction to Mobile Price Classification

Mobile price classification is a machine learning task that involves predicting the price range of mobile phones based on their features. The dataset typically includes various specifications such as battery power, camera megapixels, RAM, and other attributes that can influence the pricing of mobile devices. This guide will walk you through the process of building a classification model using Python, specifically leveraging libraries like Pandas, Scikit-learn, and AWS SageMaker for deployment.

Step 1: Setting Up Your Environment

Before starting your project, ensure you have the following:

1. **Python Installed**: Make sure you have Python installed on your system. You can download it from python.org.
2. **Required Libraries**: Install necessary libraries using pip:

```
pip install pandas scikit-learn matplotlib seaborn
```

3. **AWS Account**: If you plan to deploy your model using AWS SageMaker, create an AWS account and set up IAM permissions.

Step 2: Data Preparation

1. **Load the Dataset**: Use Pandas to load your dataset.

```
import pandas as pd

data = pd.read_csv('mobile_price_data.csv')
```

4. **Explore the Data**: Check for missing values and understand the distribution of features.

```
print(data.info())
print(data.describe())
```

6. **Feature Engineering**: Create new features if necessary or transform existing ones to improve model performance.
7. **Split the Data**: Divide your dataset into training and testing sets.

```
from sklearn.model_selection import train_test_split

X = data.drop('price_range', axis=1)
y = data['price_range']

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Step 3: Model Selection and Training

1. **Choose a Model**: For this classification task, several models can be used including Random Forests, K-Nearest Neighbors (KNN), or Decision Trees.
2. **Train the Model**:

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)
```

6. **Evaluate the Model**:

```
from sklearn.metrics import accuracy_score

predictions = model.predict(X_test)
accuracy = accuracy_score(y_test, predictions)

print(f'Model Accuracy: {accuracy * 100:.2f}%')
```

Step 4: Hyperparameter Tuning

To improve model performance further:

1. Use Grid Search for hyperparameter tuning.

```
from sklearn.model_selection import GridSearchCV

param_grid = {
 'n_estimators': [50, 100],
 'max_depth': [None, 10, 20]
}
```

```
8.
9. grid_search = GridSearchCV(RandomForestClassifier(), param_grid)
10.  grid_search.fit(X_train, y_train)
11.
   print(f'Best Parameters: {grid_search.best_params_}')
```

Step 5: Deployment Using AWS SageMaker

1. **Create an S3 Bucket**: Store your trained model in an S3 bucket.
2. **Deploy with SageMaker**:
   - Create a SageMaker endpoint to serve predictions.
   - Use `boto3` library in Python to interact with AWS services.

```
import boto3

sagemaker_client = boto3.client('sagemaker')

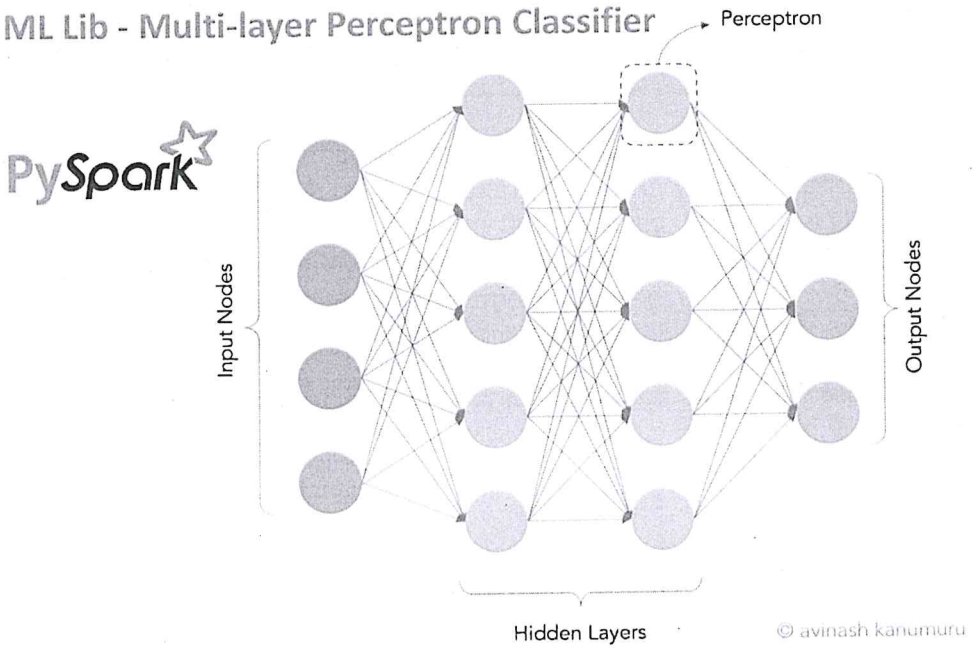
# Code for creating a SageMaker endpoint goes here...
```

3. **Test Your Endpoint** by sending sample data and receiving predictions.
4. **Delete Endpoint After Testing** to avoid incurring additional costs.

```
sagemaker_client.delete_endpoint(EndpointName='your-endpoint-name')
```

By following these steps systematically—from setting up your environment to deploying your model—you can successfully classify mobile phone prices using machine learning techniques in Python.

18.Perceptron IRIS classification



Perceptron IRIS Classification

The perceptron is a type of artificial neural network that serves as a fundamental building block for more complex models. It is particularly useful for binary classification tasks, such as distinguishing between different species of flowers in the Iris dataset. The Iris dataset, introduced by R.A. Fisher in 1936, contains measurements of iris flowers from three species: Setosa, Versicolor, and Virginica. For the purpose of this explanation, we will focus on classifying only two species: Setosa and Versicolor.

1. Understanding the Dataset

The Iris dataset consists of four features for each flower sample:

- Sepal Length
- Sepal Width
- Petal Length
- Petal Width

In our case, we will use only two features: petal length and sepal length to simplify the classification task. Each species has 50 samples, resulting in a total of 150 samples.

2. Data Preparation

To prepare the data for training the perceptron model:

- We load the dataset using libraries such as Pandas.
- We extract the relevant features (sepal length and petal length) and their corresponding labels (Setosa or Versicolor).

For example, using Python code:

```
import pandas as pd

# Load the dataset
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None)

# Extract features and labels
X = df.iloc[0:100, [0, 2]].values  # First 100 samples (Setosa and
Versicolor)
y = df.iloc[0:100, 4].values        # Corresponding labels
```

Next, we convert these labels into numerical values where Setosa is represented by -1 and Versicolor by +1:

```
y = np.where(y == 'Iris-setosa', -1, 1)
```

3. Implementing the Perceptron Algorithm

The perceptron algorithm involves initializing weights for each feature and iteratively updating these weights based on prediction errors during training.

Here's a simplified version of how the perceptron class can be implemented:

```
class Perceptron:
    def __init__(self, rate=0.01, niter=10):
        self.rate = rate
        self.niter = niter

    def fit(self, X, y):
        self.weight = np.zeros(1 + X.shape[1])
        self.errors = []

        for _ in range(self.niter):
            err = 0
            for xi, target in zip(X, y):
                update = self.rate * (target - self.predict(xi))
                self.weight[1:] += update * xi
                self.weight[0] += update
                err += int(update != 0.0)
            self.errors.append(err)

    def net_input(self, X):
        return np.dot(X, self.weight[1:]) + self.weight[0]

    def predict(self, X):
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

4. Training the Model

Once we have defined our perceptron class:

- We create an instance of it.
- We call the `fit` method with our feature matrix `X` and label vector `y`.

```
ppn = Perceptron(rate=0.01, niter=10)
ppn.fit(X, y)
```

5. Evaluating Performance

After training the model:

- We can evaluate its performance by checking how many misclassifications occurred during training epochs.
- A plot can be generated to visualize errors over epochs.

```
import matplotlib.pyplot as plt

plt.plot(range(1,len(ppn.errors)+1), ppn.errors)
plt.xlabel('Epochs')
plt.ylabel('Number of misclassifications')
plt.title('Perceptron Learning')
plt.show()
```

6. Visualizing Decision Boundaries

Finally, we can visualize how well our perceptron has learned to classify the two species by plotting decision boundaries based on learned weights:

```
x_min = X[:, 0].min() - 1
x_max = X[:, 0].max() + 1
y_min = X[:, 1].min() - 1
y_max = X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min,x_max,.01),
np.arange(y_min,y_max,.01))
Z = ppn.predict(np.array([xx.ravel(), yy.ravel()]).T).reshape(xx.shape)

plt.contourf(xx, yy, Z , alpha=0.8)
plt.scatter(X[:50, 0], X[:50 , 1], color='red', marker='o', label='setosa')
plt.scatter(X[50:100 , 0], X[50:100 , 1], color='blue', marker='x',
label='versicolor')
plt.xlabel('Sepal Length')
plt.ylabel('Petal Length')
plt.legend(loc='upper left')
plt.title('Perceptron Decision Boundary')
plt.show()
```

This visualization helps us see how effectively our perceptron model has classified Setosa and Versicolor based on their sepal and petal lengths.

In summary **the perceptron model successfully classifies iris flower species based on selected features**, demonstrating its utility in simple binary classification tasks

19.Implementation of Naive Bayes in Python – Machine Learning

To implement the Naive Bayes classifier in Python, you can use the Scikit-learn library, which provides a straightforward way to apply this algorithm. Here's a step-by-step guide:

1. **Import Libraries**: Start by importing necessary libraries.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
```

6. **Load Dataset**: Load your dataset into a Pandas DataFrame. For example, using the Iris dataset:

```
# Load the Iris dataset
data = pd.read_csv('iris.csv')
```

8. **Preprocess Data**: Split the data into features and labels.

```
X = data.iloc[:, :-1].values  # Features (all columns except last)
y = data.iloc[:, -1].values   # Labels (last column)
```

10. **Split Data**: Divide the dataset into training and testing sets.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

11. **Create Model**: Instantiate the Naive Bayes classifier.

```
model = GaussianNB()
```

12. **Train Model**: Fit the model to the training data.

```
model.fit(X_train, y_train)
```

13. **Make Predictions**: Use the trained model to make predictions on the test set.

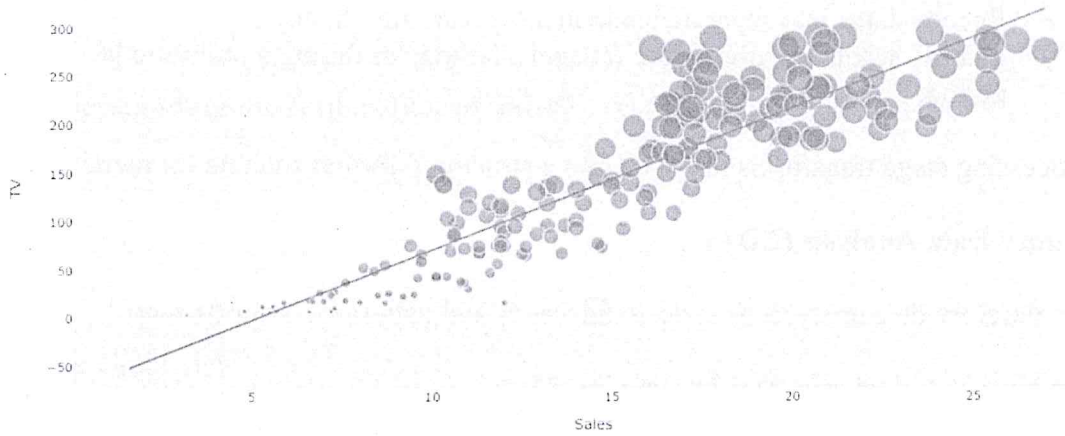
```
y_pred = model.predict(X_test)
```

14. **Evaluate Model**: Assess the performance of your model using accuracy and other metrics.

```
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test,
y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

This implementation covers loading a dataset, preprocessing it for modeling, training a Naive Bayes classifier using Scikit-learn's `GaussianNB`, making predictions on unseen data, and evaluating its performance.

20. Future Sales Prediction using Python



Future Sales Prediction using Python

Sales prediction is a crucial aspect of business strategy, allowing companies to estimate future sales based on historical data and various influencing factors. This process can be effectively implemented using Python, leveraging machine learning techniques to create accurate forecasting models. Below is a detailed step-by-step guide on how to build a sales forecasting model using Python.

1. Understanding the Importance of Sales Forecasting

Sales forecasting involves predicting future sales volumes based on past sales data and other relevant variables such as seasonality, economic conditions, and promotional events. Accurate forecasts enable businesses to make informed decisions regarding inventory management, budgeting, staffing, and marketing strategies.

2. Data Collection

The first step in building a sales forecasting model is to gather historical sales data. This dataset should ideally span several years to capture trends and seasonal patterns. The data typically includes:

- Date of sale
- Sales volume (traffic)
- Additional features such as holidays or special events

For example, you might collect data from a CSV file containing daily sales figures over the past five years.

3. Data Preprocessing

Once the data is collected, it needs to be preprocessed for analysis:

- **Cleaning the Data:** Remove any missing or erroneous entries.
- **Feature Engineering:** Create new features that may help improve the model's accuracy. For instance:
    - Extract day of the week from the date.
    - Encode dates into separate components (year, month, day).
    - Identify whether a given date falls on a holiday or during a promotional period.

This preprocessing stage transforms raw data into a structured format suitable for modeling.

4. Exploratory Data Analysis (EDA)

Conduct exploratory data analysis to understand trends and patterns in your dataset:

- Visualize historical sales trends using line plots.
- Analyze seasonal effects by comparing sales across different months or quarters.
- Identify correlations between different features (e.g., do holidays significantly boost sales?).

Tools like Matplotlib and Seaborn can be used for visualization purposes.

5. Model Selection

Choose an appropriate machine learning model for forecasting. Common models include:

- **Linear Regression:** A simple yet effective method for capturing linear relationships.
- **Random Forest Regression:** A robust ensemble method that can handle non-linear relationships well.
- **Long Short-Term Memory (LSTM):** A type of recurrent neural network suited for time series forecasting due to its ability to remember long-term dependencies.

In this case, we will focus on using LSTM due to its effectiveness in handling sequential data like time series.

6. Model Training

Split your dataset into training and testing sets—typically using the first four years of data for training and the last year for testing:

```
from sklearn.model_selection import train_test_split

# Assuming 'data' is your preprocessed DataFrame
train_data = data[data['date'] < '2020-01-01']
test_data = data[data['date'] >= '2020-01-01']
```

Next, normalize your input features so that they are within a similar scale, which helps improve model performance:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

```
train_scaled = scaler.fit_transform(train_data)
```

Then define your LSTM model architecture using Keras:

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

model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_input_features, 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Train the model with your training dataset:

```
model.fit(train_scaled, epochs=100)
```

7. Model Evaluation

After training the model, evaluate its performance on the test set by making predictions and comparing them against actual values:

```
predictions = model.predict(test_scaled)
```

Use metrics such as Mean Absolute Error (MAE) or Root Mean Squared Error (RMSE) to quantify accuracy.

8. Making Future Predictions

Once satisfied with your model's performance, you can use it to make future predictions by feeding it new input data formatted similarly to your training set.

```
future_predictions = model.predict(new_data_scaled)
```

These predictions can then be denormalized back into original units if necessary.

9. Deployment

Finally, deploy your trained model into production where it can continuously receive new input data and provide ongoing forecasts as part of business operations.

By following these steps systematically—from understanding the importance of sales forecasting through collecting and processing data to selecting models and evaluating their performance—you can effectively implement future sales prediction using Python.