| **Ex.no:** | **Data Pre-processing using Pandas** |
| --- | --- |
| **Date:** |

**AIM:**

To Load Real Time data Set and Python Libraries, Installing Libraries through Anaconda Prompt, Perform data pre-processing through Pandas Library.

**ALGORITHM:**

Step 1: Start the code

Step 2: Open a Jupyter Notebook

Step 3: Install the needed python packages using Pip install package name.

Step 4: Import the needed python packages using Import package name.

import pandas as pd, import numpy as np, import matplotlib.pyplot as plt.

Step 5: Import the data set using pandas package.

df = pd.read\_csv("D:\Jeyashri\IBM\Datasets\Social\_Network\_Ads.csv")

df

Step 6: Drop the null values by using pandas method.

df = df.dropna()

df

Step 7: Clear the Duplicate data by using pandas method.

df = df.drop\_duplicates()

df

Step 8: Create new dummies values to process the data based on column.

df = pd.get\_dummies(df, columns=['Gender'])

df

Step 10: Stop the process.

**PROGRAM:**

import pandas as pd

#Uploading data into dataframe

df = pd.read\_csv("D:\Jeyashri\IBM\Datasets\Social\_Network\_Ads.csv")

df

# Fill missing values with a specific value

df = df.fillna(values)

df

# Drop rows with missing values

df = df.dropna()

df

# Interpolate missing values

df = df.interpolate()

df

# Remove duplicate rows

df = df.drop\_duplicates()

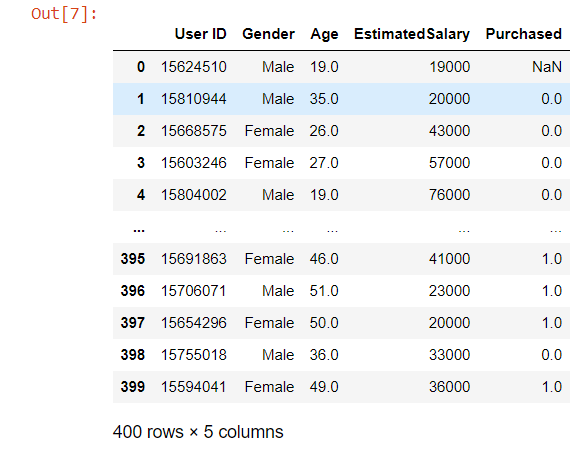
df

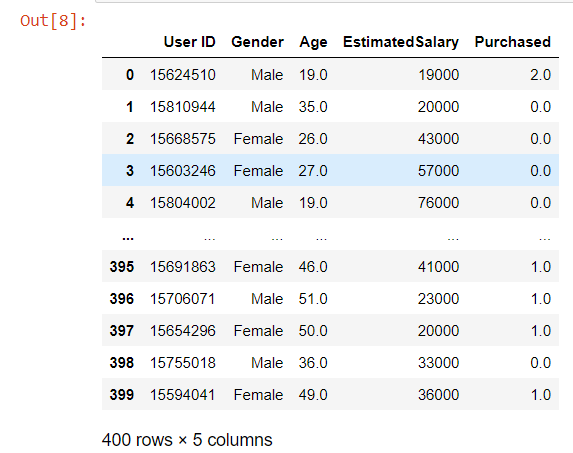
# One-hot encoding categorical columns

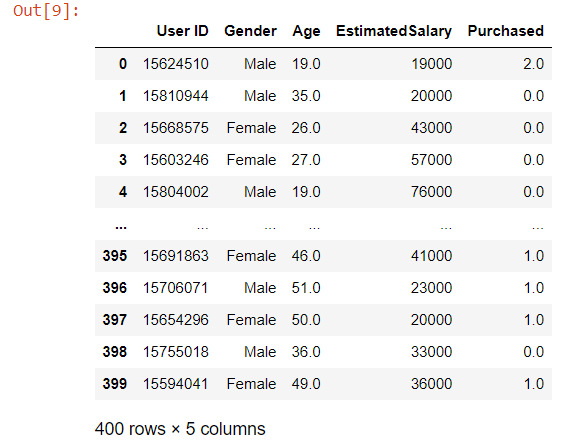
df = pd.get\_dummies(df, columns=['Gender'])

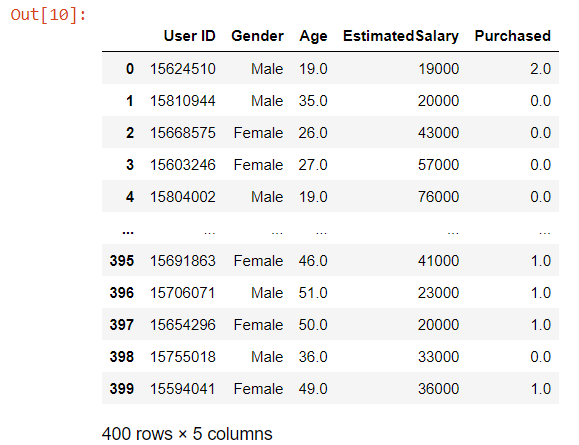
df

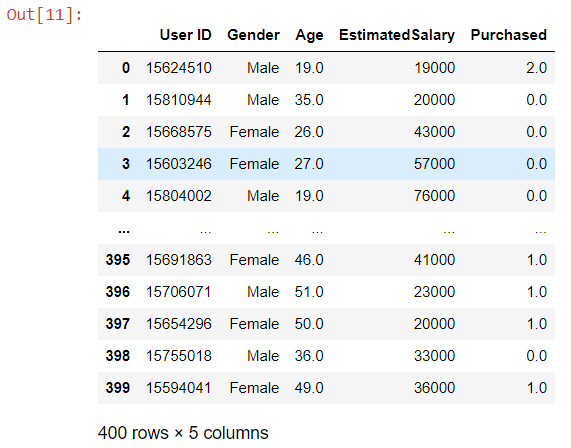
**OUTPUT:**

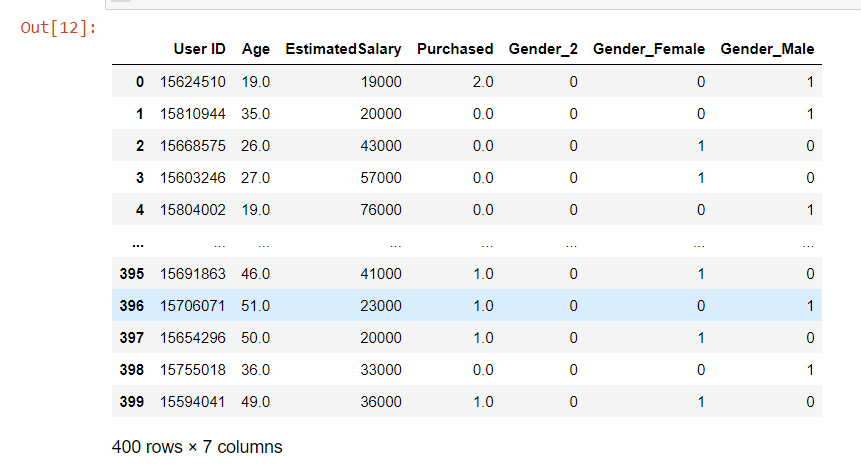
****

****

****

****

****

****

**RESULT:**

Hence the data pre-process successfully.

| **Ex.no:** | **Bayesian network** |
| --- | --- |
| **Date:** |

**AIM:**

To Construct a Bayesian network considering student data.

**ALGORITHM:**

Step 1: Start the code

Step 2: Open a Jupyter Notebook

Step 3: Install the needed python packages using Pip install package name.

Step 4: Import the needed python packages using Import package name.

import pandas as pd, import numpy as np, import matplotlib.pyplot as plt.

from sklearn.naive\_bayes import GaussianNB

Step 5: Import a data set by using pandas

dataset = pd.read\_csv("results.csv")

Step 6: Select the x and y data points with the help of loc() methods.

X = data.loc[:, ["Hours", "StudentId"]] # Features

y = data["Result"] # Target

new=pd.DataFrame(X,y)

Step 7: Split the dataset into train and test for training and testing the machine.

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.20, random\_state = 0)

Step 8: Use GaussianNB Algorithm to fit and predict the model

from sklearn.naive\_bayes import GaussianNB

classifier = GaussianNB()

classifier.fit(X\_train, y\_train)

Step 9: Finally Evaluate the result using metrics ()

from sklearn.metrics import confusion\_matrix,accuracy\_score

cm = confusion\_matrix(y\_test, y\_pred)

ac = accuracy\_score(y\_test,y\_pred)

cr= classification\_report(y\_test, y\_pred))

print(cm) print(ac) print(cr)

Step 10: Visualise the confusion-matrix by using Seaborn package

Step 11: Stop the Implementation

**PROGRAM:**

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

import matplotlib.pyplot as plt

import seaborn as sns

# Load data from CSV file

data = pd.read\_csv("results.csv")

print(data)

# Assuming the last column is the target column and rest are features

X = data.loc[:, ["Hours", "StudentId"]] # Features

y = data["Result"] # Target

new=pd.DataFrame(X,y)

print(new)

# Split data into train and test 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)

# Initialize Naive Bayes classifier

classifier = GaussianNB()

classifier.fit(X\_train, y\_train)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

# Classification report

print("Classification Report:")

print(classification\_report(y\_test, y\_pred))

# Calculate false percentage

false\_percentage = (1 - accuracy) \* 100

print("False Percentage:", false\_percentage)

new=pd.DataFrame(X\_test,y\_pred)

print(new)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

conf\_matrix

# Plot confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(conf\_matrix, annot=True, fmt="d", cmap="Blues", cbar=False)

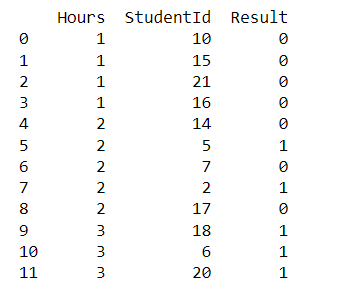
plt.title("Confusion Matrix")

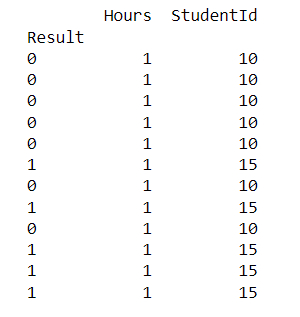
plt.xlabel("Predicted Labels")

plt.ylabel("True Labels")

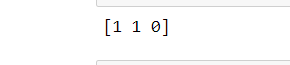
plt.show()

**OUTPUT:**

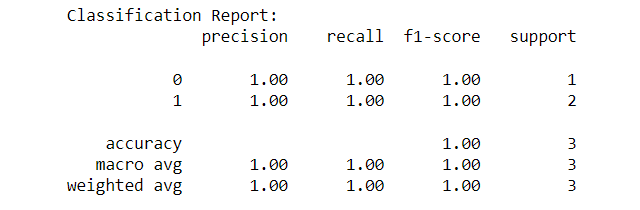
****

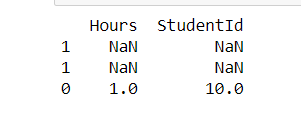
****

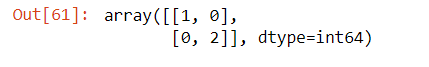
****

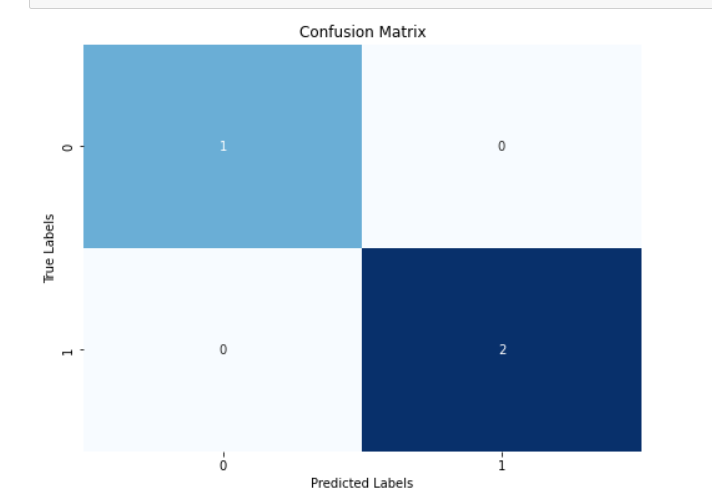
****

****

****

****

****

****

**RESULT:**

Hence the Student dataset evaluated successfully with help of Bayesian network

| **Ex.no:** | **K-Means Clustering** |
| --- | --- |
| **Date:** |

**AIM:**

To Implement a K-Means Clustering.

**ALGORITHM:**

Step 1: Start the code

Step 2: Open a Jupyter Notebook

Step 3: Install the needed python packages using Pip install package name.

Step 4: Import the needed python packages using Import package name.

import pandas as pd, import numpy as np, import matplotlib.pyplot as plt.

from sklearn.cluster import KMeans

Step 5: Import a data set by using pandas

data = pd.read\_csv("D:\\Jeyashri\\IBM\\Datasets\\Country clusters.csv")

data

Step 6: Scatter the data point to find out the cluster formation by using matplot.pyplot package.

Step 7: Select the x data points with the help of iloc() methods

x = data.iloc[:,1:3]

x

Step 8: Select the model and define number of cluster to be form.

kmeans = KMeans(3)

kmeans.fit(x)

Step 9: Use the fit\_predict() method to perform both the fit and predict in a single line. And store the X value in Identified\_clusters variable.

identified\_clusters = kmeans.fit\_predict(x)

identified\_clusters

Step 10: Make a copy on the data set using copy () method and for a cluster.

Step 11: Stop the code

**PROGRAM:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

data = pd.read\_csv("D:\\Jeyashri\\IBM\\Datasets\\Country clusters.csv")

data

plt.scatter(data['Longitude'],data['Latitude'])

plt.xlim(-180,180)

plt.ylim(-90,90)

plt.xlabel("Longitude")

plt.ylabel("Latitude")

plt.show()

x = data.iloc[:,1:3]

x

kmeans = KMeans(3)

kmeans.fit(x)

identified\_clusters = kmeans.fit\_predict(x)

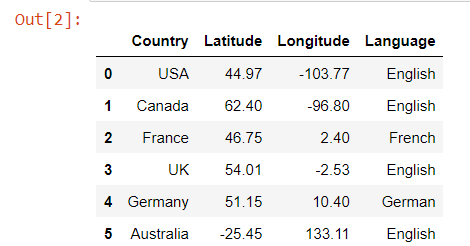
identified\_clusters

data\_with\_clusters = data.copy()

data\_with\_clusters['Clusters'] = identified\_clusters

plt.scatter(data\_with\_clusters['Longitude'],data\_with\_clusters['Latitude'],c=data\_with\_clusters['Clusters'],cmap='rainbow')

**OUTPUT:**

****

**Table

Description automatically generated**

**Graphical user interface, application

Description automatically generated**

****

**Chart, scatter chart

Description automatically generated**

**RESULT:**

Hence the Clusters are formed successfully.

| **Ex.no:** | **ID3 Algorithm** |
| --- | --- |
| **Date:** |

**AIM:**

To Implement ID3 algorithm with the help of use define functions

**ALGORITHM:**

Step 1: Start the code

Step 2: Open a Jupyter Notebook

Step 3: Install the needed python packages using Pip install package name.

Step 4: Import the needed python packages using Import package name.

import math, import pandas as pd, import numpy as np, import matplotlib.pyplot as plt and from operator import itemgetter

Step 5: Create a decision tree class to perform user define function using ID3 algorithm

Step 6: Create a Contractor to init the process by set u the target, positive, parent\_val, parent

def \_\_init\_\_(self, df, target, positive, parent\_val, parent):

Step 7: Create user define function like entropy, Gain, splitter, etc to process the descision tree

Step 8: Import a data set by using pandas.

df = pd.read\_excel('exa.xlsx')

df

Step 8: Updated note values by using update node method.

Step 9: Print the values Parent values using print\_tree(dt)

Step 10: Save and stop the process.

**PROGRAM:**

import math

import pandas as pd

class DecisionTree:

def \_\_init\_\_(self, df, target, positive, parent\_val, parent):

self.data = df

self.target = target

self.positive = positive

self.parent\_val = parent\_val

self.parent = parent

self.childs = []

self.decision = ''

def \_get\_entropy(self, data):

p = sum(data[self.target] == self.positive)

n = data.shape[0] - p

p\_ratio = p / (p + n)

n\_ratio = 1 - p\_ratio

entropy\_p = -p\_ratio \* math.log2(p\_ratio) if p\_ratio != 0 else 0

entropy\_n = -n\_ratio \* math.log2(n\_ratio) if n\_ratio != 0 else 0

return entropy\_p + entropy\_n

def \_get\_gain(self, feat):

avg\_info = sum(self.\_get\_entropy(self.data[self.data[feat] == val]) \*

sum(self.data[feat] == val) / self.data.shape[0]

for val in self.data[feat].unique())

return self.\_get\_entropy(self.data) - avg\_info

def \_get\_splitter(self):

self.splitter = max(self.gains, key=lambda x: x[1])[0]

def update\_nodes(self):

self.features = [col for col in self.data.columns if col != self.target]

self.entropy = self.\_get\_entropy(self.data)

if self.entropy != 0:

self.gains = [(feat, self.\_get\_gain(feat)) for feat in self.features]

self.\_get\_splitter()

residual\_columns = [k for k in self.data.columns if k != self.splitter]

for val in self.data[self.splitter].unique():

df\_tmp = self.data[self.data[self.splitter] == val][residual\_columns]

tmp\_node = DecisionTree(df\_tmp, self.target, self.positive, val, self.splitter)

tmp\_node.update\_nodes()

self.childs.append(tmp\_node)

def print\_tree(node, depth=0):

if node:

print(f"{' ' \* depth}Parent: {node.parent} | Parent Value: {node.parent\_val}")

for child in node.childs:

print\_tree(child, depth + 1)

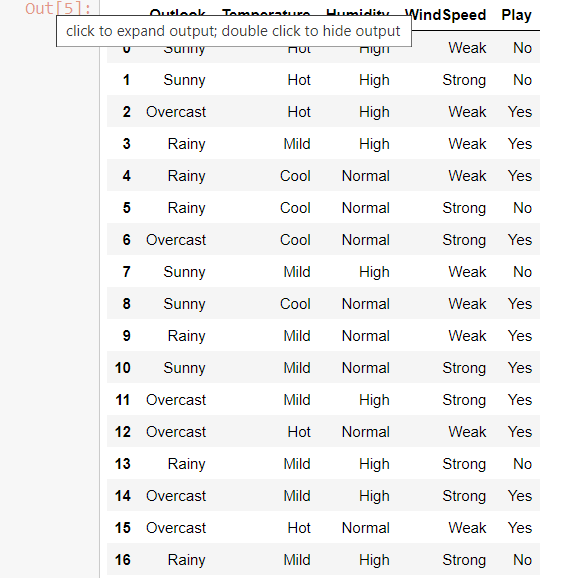
df = pd.read\_excel('exa.xlsx')

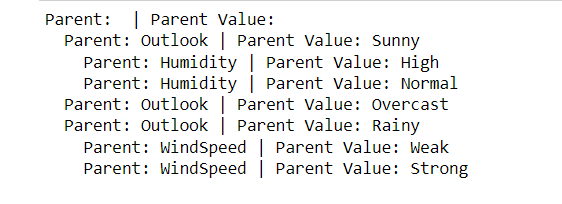
dt = DecisionTree(df, 'Play', 'Yes', '', '')

dt.update\_nodes()

print\_tree(dt)

**OUTPUT:**



****

**RESULT:**

Hence the new nodes have been creating using ID3 Algorithm successfully.

| **Ex.no:** | **Non- Parametric Locally Weighted Regression** |
| --- | --- |
| **Date:** |

**AIM:**

To Implement Non- Parametric Locally Weighted Regression and Visualize the value

**ALGORITHM:**

Step 1: Start the code

Step 2: Open a Jupyter Notebook

Step 3: Install the needed python packages using Pip install package name.

Step 4: Import the needed python packages using Import package name.

import math, import pandas as pd, import numpy as np, import matplotlib.pyplot as plt and from operator import itemgetter

Step 5: Create a pre-define function calculating the weight of regression

def locally\_weighted\_regression(test\_point, X, y, tau):

test\_point: The point at which prediction is to be made.

X: Feature matrix.

y: Target values.

tau: Bandwidth parameter for weighting.

Step 6: Calculate weights for each training point based on their distance from the test point using a Gaussian kernel.

Step 7: Call the plot\_lwr function with the generated dataset X and y, along with the specified tau\_values

Step 8: Generate a random dataset X of 100 points between 0 and 5. Calculate y values by taking the sine of each X value.

Step 9: Visualize the data using plot method.

Step 10: Stop the process.

**PROGRAM:**

import numpy as np

import matplotlib.pyplot as plt

def locally\_weighted\_regression(test\_point, X, y, tau):

# Locally Weighted Regression (LWR) function

m = X.shape[0]

weights = np.exp(-np.sum((X - test\_point)\*\*2, axis=1) / (2 \* tau\*\*2))

W = np.diag(weights)

theta = np.linalg.inv(X.T @ W @ X) @ (X.T @ W @ y)

prediction = test\_point @ theta

return prediction

def plot\_lwr(X, y, tau\_values):

#Plotting the Locally Weighted Regression predictions for different tau values

X\_test = np.linspace(np.min(X), np.max(X), 100).reshape(-1, 1)

plt.figure(figsize=(10, 6))

plt.scatter(X, y, color='blue', label='Data points')

for tau in tau\_values:

predictions = [locally\_weighted\_regression(np.array([x]), X, y, tau) for x in X\_test]

plt.plot(X\_test, predictions, label=f'tau={tau}')

plt.xlabel('X')

plt.ylabel('y')

plt.title('Locally Weighted Regression')

plt.legend()

plt.grid(True)

plt.show()

# Generate sample dataset

np.random.seed(0)

X = np.sort(5 \* np.random.rand(100, 1), axis=0)

y = np.sin(X).ravel()

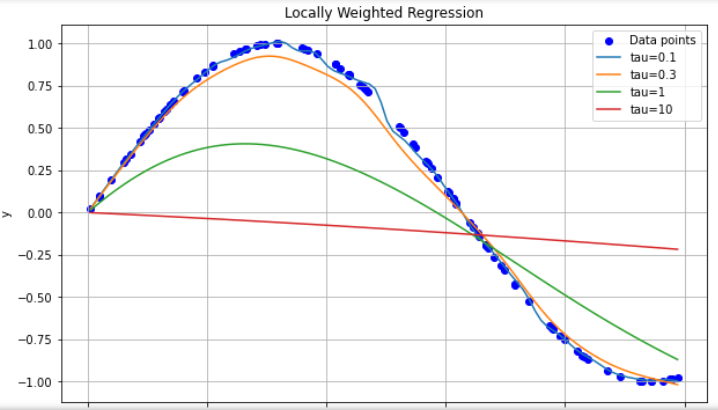
# Define tau values

tau\_values = [0.1, 0.3, 1, 10]

# Plot Locally Weighted Regression for different tau values

plot\_lwr(X, y, tau\_values)

**OUTPUT:**

****

**RESULT:**

Hence the above Non- Parametric Locally Weighted Regression executed successful

| **Ex.no:** | **k-Nearest Neighbour** |
| --- | --- |
| **Date:** |

**AIM:**

To Implement k-Nearest Neighbour algorithm to classify the iris data set.

**ALGORITHM:**

Step 1: Start the code

Step 2: Open a Jupyter Notebook

Step 3: Install the needed python packages using Pip install package name.

Step 4: Import the needed python packages using Import package name.

import pandas as pd, import numpy as np, import matplotlib.pyplot as plt.

from sklearn.neighbors import KNeighborsClassifier

Step 5: Import a data set by using pandas

data= pd.read\_csv(head.csv')

Step 6: Convert the dataset into a pandas data frame.

dataset= pd.DataFrame(data)

Step 7: Select the x and y data points with the help of iloc() methods.

X = dataset.iloc[:, [1, 2]].values

y = dataset.iloc[:, -1].values

Step 8: Split the dataset into train and test for training and testing the machine.

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state = 0)

Step 9: Train and test the machine by using fit () and predit() method.

classifier.fit(X\_train, y\_train)

y\_pred = classifier.predict(X\_test)

Step 10: Finally Evaluate the result using metrics ()

from sklearn.metrics import accuracy\_score

Step 11: Create new two variable to store the count of correct-predictions and Wrong-predictions values.

Step 12: Calculate the value using length of y-test and y-pred data

Step 13: Stop the Implementation.

**PROGRAM:**

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score

# Load dataset

dataset = pd.read\_csv('head.csv')

# Extracting features and target variable

X = dataset.iloc[:, [1, 2]].values

y = dataset.iloc[:, -1].values

# Split dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.20, random\_state=0)

# Initialize k-NN classifier

k = 3 # You can adjust the value of k

knn = KNeighborsClassifier(n\_neighbors=k)

# Train the classifier

knn.fit(X\_train, y\_train)

# Predict classes for the test set

y\_pred = knn.predict(X\_test)

# Compute accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy}")

# Print correct and wrong predictions

correct\_predictions = 0

wrong\_predictions = 0

for i in range(len(y\_test)):

if y\_test[i] == y\_pred[i]:

print(f"Correct prediction: Predicted {y\_pred[i]}, Actual {y\_test[i]}")

correct\_predictions += 1

else:

print(f"Wrong prediction: Predicted {y\_pred[i]}, Actual {y\_test[i]}")

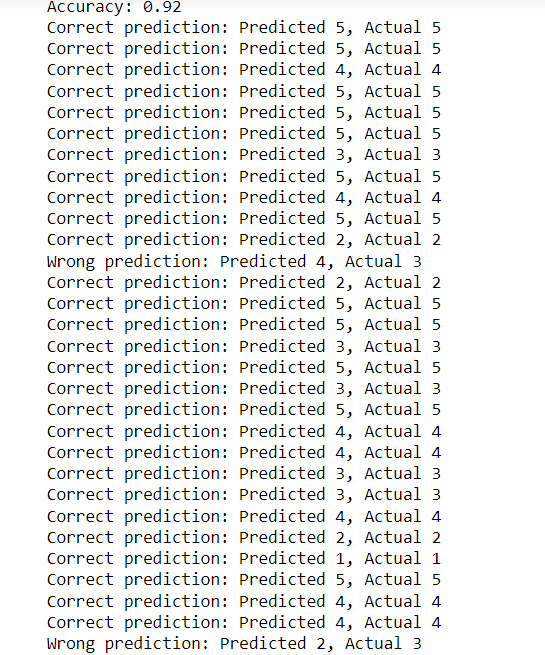
wrong\_predictions += 1

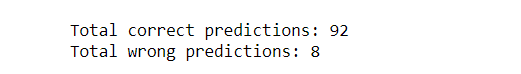
print(f"\nTotal correct predictions: {correct\_predictions}")

print(f"Total wrong predictions: {wrong\_predictions}")

**OUTPUT:**

****

****

****

**RESULT:**

This the above dataset has been evaluated successfully and find out the count of correct and wrong prediction.

| **Ex.no:** | **Semi Supervised Classifier** |
| --- | --- |
| **Date:** |

**AIM:**

To Assuming a set of documents that need to be classified, use the Semi Supervised Classifier model to perform this task. Built-in Java classes/API can be used to write the program. Calculate the accuracy, precision, and recall for your data set.

**ALGORITHM:**

Step 1: Start the code

Step 2: Open a Jupyter Notebook

Step 3: Install the needed python packages using Pip install package name. Step 4: Import the needed python packages using Import package name.

Step 5: Prepare a labeled and unlabeled data for implementation

Step 6: Combine data for extraction feature from the input data.

Step 7: Create label distributions matrix

Step 8: Split labeled data into training set and test set

Step 9: Using predict method predict the new value

Step 10: Evaluate the output using metric method

Step 11: Stop the process

**PROGRAM:**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.semi\_supervised import LabelPropagation

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score

from sklearn.model\_selection import train\_test\_split

import numpy as np

# Sample data (replace with your actual data)

labeled\_data = [("This is a document about sports", "Sports"),

("This is a news article", "News"),

("Another document about sports", "Sports"),

("A text sample about politics", "Politics"),

("A document discussing music", "Music")]

unlabeled\_data = ["This document discusses machine learning",

"Another document about music",

"A short text sample"]

# Combine data for feature extraction

all\_data = [text for text, \_ in labeled\_data] + unlabeled\_data

# Extract labels from labeled data

texts, labels = zip(\*labeled\_data)

# Feature Extraction (TF-IDF)

vectorizer = TfidfVectorizer(max\_features=500)

features = vectorizer.fit\_transform(all\_data)

# Convert features to dense numpy array

features\_dense = features.toarray()

# Get all unique labels

all\_labels = sorted(set(labels))

# Create label distributions matrix

label\_distributions = np.zeros((len(texts), len(all\_labels)))

for i, label in enumerate(labels):

label\_distributions[i, all\_labels.index(label)] = 1

# Split labeled data into training set and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(features\_dense[:len(texts)], labels, test\_size=0.2, random\_state=42)

# Convert y\_train to indices of true classes

y\_train\_indices = np.array([all\_labels.index(label) for label in y\_train])

# Train the semi-supervised classifier

semi\_clf = LabelPropagation()

semi\_clf.fit(X\_train, y\_train\_indices) # Ensure y\_train\_indices is passed as is

# Predict labels for test set

predictions = semi\_clf.predict(X\_test)

# Calculate evaluation metrics

accuracy = accuracy\_score(np.array([all\_labels.index(label) for label in y\_test]), predictions)

precision = precision\_score(np.array([all\_labels.index(label) for label in y\_test]), predictions, average='weighted', labels=np.unique(predictions))

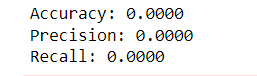
recall = recall\_score(np.array([all\_labels.index(label) for label in y\_test]), predictions, average='weighted', labels=np.unique(predictions))

print(f"Accuracy: {accuracy:.4f}")

print(f"Precision: {precision:.4f}")

print(f"Recall: {recall:.4f}")

**OUTPUT:**

****

**RESULT:**

This the above dataset has been executed successfully

| **Ex.no:** | **Implementing Q Learning with Linear Function** |
| --- | --- |
| **Date:** |

**AIM:**

To Implementing Q Learning with Linear Function

**ALGORITHM:**

Step 1: Start the code

Step 2: Open a Jupyter Notebook

Step 3: Install the needed python packages using Pip install package name.

Step 4: Import the needed python packages using Import package name.

import numpy as np.

Step 5: Create a new class called QlearningLinearFA to Implement Qlearning Algorithm.

Step 6: Initialize Q-learning with linear function approximation

Step 7: Select action using Q-learning policy for Update Q-function approximation.

Step 8: Simulate new environment, get reward and next state

Step 10: Stop the process

**PROGRAM:**

import numpy as np

class QLearningLinearFA:

def \_\_init\_\_(self, num\_features, num\_actions, learning\_rate=0.1, discount\_factor=0.9, epsilon=0.1):

self.num\_features = num\_features

self.num\_actions = num\_actions

self.learning\_rate = learning\_rate

self.discount\_factor = discount\_factor

self.epsilon = epsilon

self.weights = np.zeros((num\_actions, num\_features))

def select\_action(self, state):

if np.random.rand() < self.epsilon:

return np.random.choice(self.num\_actions) # Random action

else:

return np.argmax(np.dot(self.weights, state)) # Greedy action

def update\_weights(self, state, action, reward, next\_state):

target = reward + self.discount\_factor \* np.max(np.dot(self.weights, next\_state))

predicted = np.dot(self.weights[action], state)

error = target - predicted

self.weights[action] += self.learning\_rate \* error \* state

num\_features = 4 # Number of features representing the state

num\_actions = 3 # Number of possible actions

# Initialize Q-learning with linear function approximation

ql = QLearningLinearFA(num\_features, num\_actions)

# Example training loop

num\_episodes = 1000

for episode in range(num\_episodes):

# Simulate environment, get initial state

state = np.random.rand(num\_features)

done = False

total\_reward = 0

while not done:

# Select action using Q-learning policy

action = ql.select\_action(state)

# Simulate environment, get reward and next state

next\_state = np.random.rand(num\_features)

reward = np.random.randn() # Replace with actual reward from environment

done = np.random.rand() < 0.1 # Example termination condition

# Update Q-function approximation

ql.update\_weights(state, action, reward, next\_state)

# Update current state

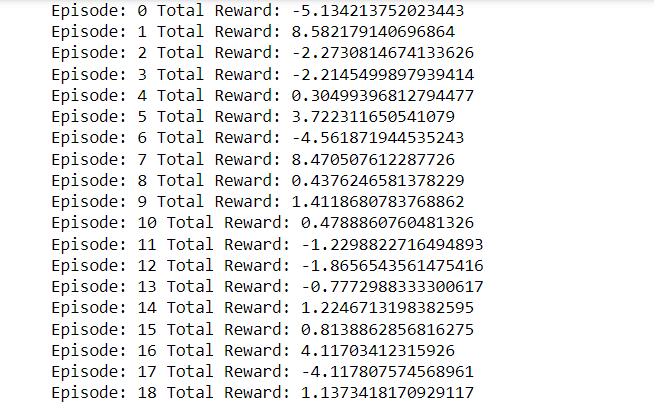
state = next\_state

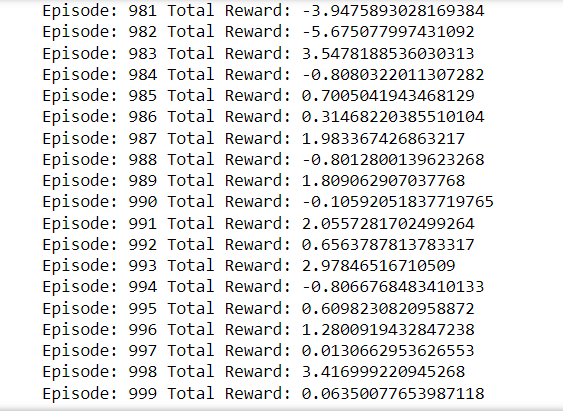
# Accumulate total reward

total\_reward += reward

print("Episode:", episode, "Total Reward:", total\_reward)

**OUTPUT:**

****

****

**RESULT:**

Hence the above program executed successfully.

| **Ex.no:** | **Implement the Policy Gradient** |
| --- | --- |
| **Date:** |

**AIM:**

To Implement the Policy Gradient

**ALGORITHM:**

Step 1: Start the code

Step 2: Open a Jupyter Notebook

Step 3: Install the needed python packages using Pip install package name.

Step 4: Import the needed python packages using Import package name.

import numpy as np.

Step 5: Create class called REINFORCEBaselineAgent

Step 6: Create a constructer for processing Time series

Step 7: Create a loop for tarin the data to the model

num\_states = 5

num\_actions = 2

num\_episodes = 1000

Step 8: Create class called ActorCriticBaselineAgent for basing the base line

Step 9: print the Episode values

print("Actor-Critic Episode {}: Total Reward = {}".format(episode + 1, episode\_rewards))

Step 10: Stop the process

**PROGRAM:**

import numpy as np #package

class REINFORCEAgent:# class

def \_\_init\_\_(self, num\_actions, num\_states, gamma=0.99, learning\_rate=0.01):

# gamma is discount factor for finding future reward

self.num\_actions = num\_actions

self.num\_states = num\_states

self.gamma = gamma

self.learning\_rate = learning\_rate

self.policy = np.zeros((num\_states, num\_actions))

def get\_action(self, state): #return action for current policy

action\_probs = self.\_softmax(self.policy[state])

#Accesses the policy for the current state

#softmax function that takes as input a vector of real numbers and returns a vector of probabilities

return np.random.choice(self.num\_actions, p=action\_probs)

def train(self, episode):

states, actions, rewards = zip(\*episode)

returns = self.\_calculate\_returns(rewards)

for t, (state, action) in enumerate(zip(states, actions)):

delta = returns[t] - self.policy[state, action]

self.policy[state, action] += self.learning\_rate \* delta

print("Episode training complete.")

#enumerate built-in function that allows you to loop over an iterable (such as a list, tuple, or string)

def \_calculate\_returns(self, rewards):

G = 0

returns = []

for r in reversed(rewards):

G = r + self.gamma \* G

returns.insert(0, G)

return returns

def \_softmax(self, x):

exp\_values = np.exp(x - np.max(x)) #exponential

return exp\_values / np.sum(exp\_values)

# Simple environment

class SimpleEnvironment:# class

def \_\_init\_\_(self, num\_states, num\_actions):

self.num\_states = num\_states

self.num\_actions = num\_actions

def reset(self):

return 0

def step(self, state, action):

new\_state = max(0, min(self.num\_states - 1, state + (action \* 2 - 1)))

reward = 1 if new\_state == self.num\_states - 1 else 0

return new\_state, reward

# Training loop

num\_states = 5

num\_actions = 2

num\_episodes = 1000

env = SimpleEnvironment(num\_states, num\_actions)

agent = REINFORCEAgent(num\_actions, num\_states)

for episode\_num in range(num\_episodes):

state = env.reset()

episode = []

total\_reward = 0 # Track total reward for the episode

done = False

while not done:

action = agent.get\_action(state)

next\_state, reward = env.step(state, action)

total\_reward += reward # Accumulate reward for the episode

episode.append((state, action, reward))

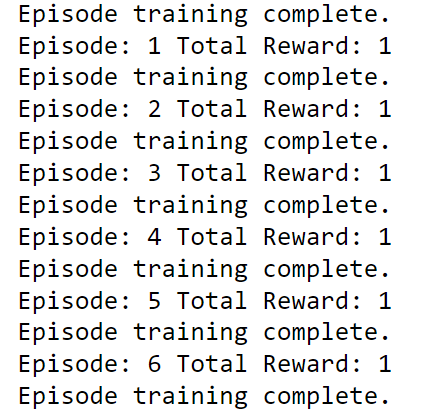
state = next\_state

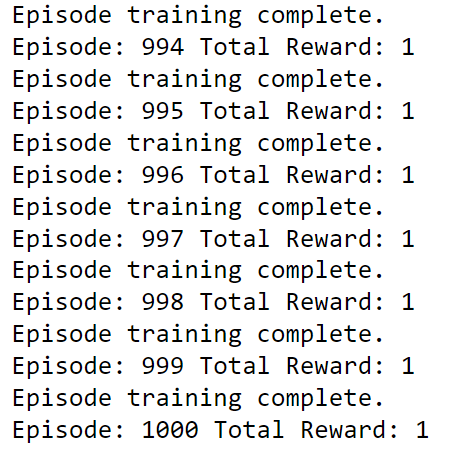
done = next\_state == num\_states - 1

agent.train(episode)

print("Episode:", episode\_num + 1, "Total Reward:", total\_reward) # Print total reward for the episode

**OUTPUT:**

****

****

**RESULT:**

Hence the above program executed successfully.

| **Ex.no:** | **Time Series Data processing** |
| --- | --- |
| **Date:** |

**AIM:**

To process Time series using decomposition methods

**ALGORITHM:**

Step 1: Start the code

Step 2: Open a Jupyter Notebook

Step 3: Install the needed python packages using Pip install package name.

Step 4: Import the needed python packages using Import package name.

import pandas as pd, import matplotlib.pyplot as plt

from statsmodels.tsa.seasonal import seasonal\_decompose

from statsmodels.tsa.stattools import adfuller

Step 5: Import the data set using pandas package.

df = pd.read\_csv("D:\Jeyashri\IBM\Datasets\Social\_Network\_Ads.csv")

df

Step 6: plot the data using matplotlib.pyplot before processing it

Step 7: Decompose the time series data for seasonal period of 12 months

decomposition = seasonal\_decompose(data[data.columns[0]], model='additive', period=12)

Step 8: Check for stationarity using Augmented Dickey-Fuller test

adf\_result = adfuller(data[data.columns[0]])

print('ADF Statistic:', adf\_result[0])

print('p-value:', adf\_result[1])

print('Critical Values:', adf\_result[4])

Step 9: Stop the process

**PROGRAM:**

import pandas as pd

import matplotlib.pyplot as plt

from statsmodels.tsa.seasonal import seasonal\_decompose

from statsmodels.tsa.stattools import adfuller

# Load time series data

data = pd.read\_csv('TimeGender.csv', index\_col=0, parse\_dates=True)

# Plot the time series data

plt.figure(figsize=(10, 6))

plt.plot(data.index, data[data.columns[0]])

plt.title('Time Series Data')

plt.xlabel('Date')

plt.ylabel('Value')

plt.show()

# Decompose the time series data

decomposition = seasonal\_decompose(data[data.columns[0]], model='additive', period=12)

# Assuming a seasonal period of 12 months

trend = decomposition.trend

seasonal = decomposition.seasonal

residual = decomposition.resid

# Plot the decomposed components

plt.figure(figsize=(10, 8))

plt.subplot(411)

plt.plot(data[data.columns[0]], label='Original')

plt.legend(loc='best')

plt.subplot(412)

plt.plot(trend, label='Trend')

plt.legend(loc='best')

plt.subplot(413)

plt.plot(seasonal, label='Seasonal')

plt.legend(loc='best')

plt.subplot(414)

plt.plot(residual, label='Residual')

plt.legend(loc='best')

plt.tight\_layout()

# Check for stationarity using Augmented Dickey-Fuller test

adf\_result = adfuller(data[data.columns[0]])

print('ADF Statistic:', adf\_result[0])

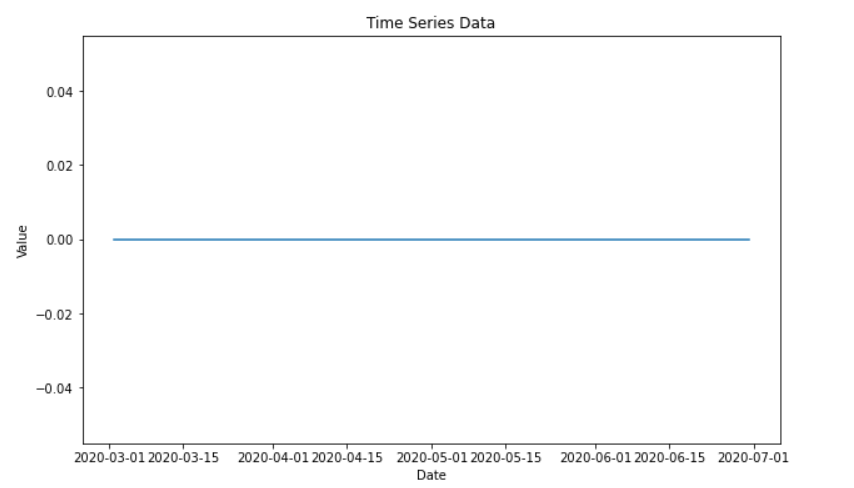
print('p-value:', adf\_result[1])

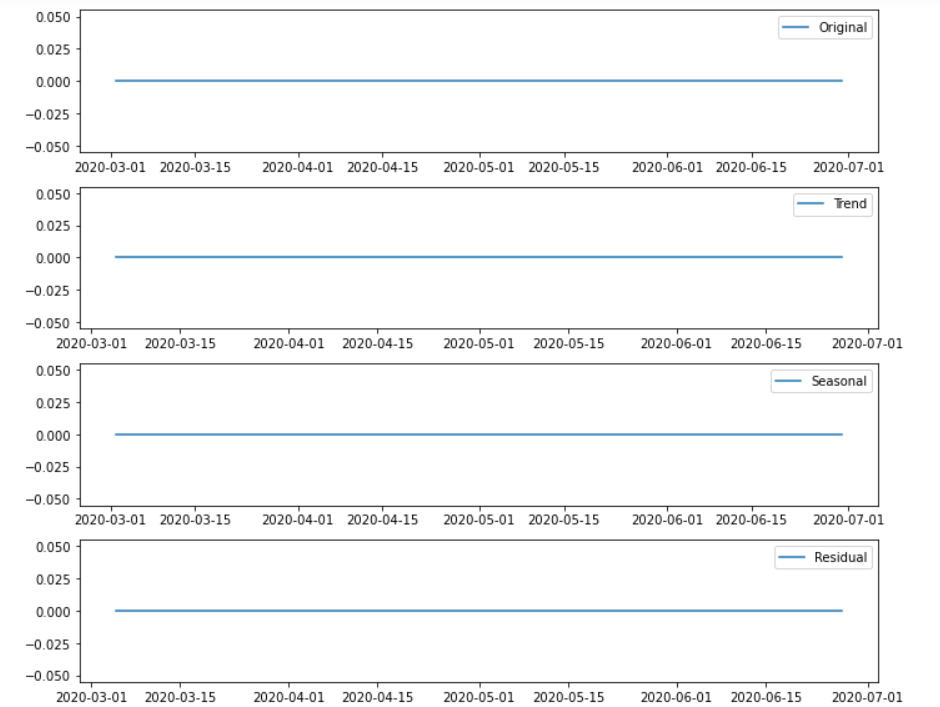
print('Critical Values:', adf\_result[4])

# Identify the trend pattern

plt.show()

**OUTPUT:**

****

****

**RESULT:**

Hence the above program executed successfully.