MEASI INSTITUTE OF INFORMATION TECHNOLOGY

(Approved by AICTE & Affiliated to University of Madras)

CHENNAI – 600 014

MEASI
Institute of
Information
Technology

MASTER OF COMPUTER APPLICATIONS

ACADEMIC YEAR 2024-2025

SEMESTER – III

Practical Record

ADVANCED MACHINE LEARNING LAB (535S3A)

REG. NO : ____________________________________

NAME : ____________________________________

BATCH : ____________________________________

MEASI INSTITUTE OF INFORMATION TECHNOLOGY

(Approved by AICTE & Affiliated to University of Madras)

CHENNAI- 600 014

MCA PRACTICALS

ADVANCED MACHINE LEARNING LAB
(535S3A)

Academic Year 2024-2025

Semester - III

NAME	**:**	**CLASS**	**:**
REG-NO	**:**	**BATCH**	**:**

This is to certify that this is the bonafide record of work done in the Computer Science Laboratory of **MEASI Institute of Information Technology**, submitted for the **University of Madras** Practical Examination held on .... at **MEASI Institute of Information Technology**, **Chennai-600 014**.

STAFF IN-CHARGE **PROFESSOR & ASSIST.DIRECTOR**

INTERNAL EXAMINER **EXTERNAL EXAMINER**

DATE: ____________________________

INDEX

S.No.	DATE	Program Name	Page No.	Signature
01		Write a python program to compute the Central Tendency Measures: Mean, Median, Mode, Measure of Dispersion: Variance, Standard Deviation.		
02		a) - Implement a Linear Regression and Multiple Linear Regression with a Real Dataset. b) - Implement a Linear Regression and Multiple Linear Regression with a Real Dataset.		
03		Implementation of Logistic Regression using sklearn.		
04		Implement a binary classification model.		
05		Classification with Nearest Neighbours and NavieBayes Algorithm.		
06		Implementation Decision tree for classification using sklearn and its parameter tuning.		
07		Implement the k-means algorithm.		
08		Implement an Image Classifier using CNN in TensorFlow/Keras.		
09		Implement an Autoencoder in TensorFlow/Keras.		
10		Implement a Simple LSTM using TensorFlow/Keras.		

Write a python program to compute the Central Tendency Measures: Mean, Median, Mode, Measure of Dispersion: Variance, Standard Deviation.

EX No: 01

Date:

Aim:

Write a python program to compute the Central Tendency Measures: Mean, Median, Mode, Measure of Dispersion: Variance, Standard Deviation.

Algorithm:

Step 1 : Start the program.

Step 2 : Import Statistics package.

Step 3 : Create a function and give the parameter as numbers.

Step 4 : In that, create Central Tendency Measures (mean, median, and mode) and Measures of Dispersion (variance and standard deviation) and then return the values as dictionary.

Step 5 : Outside the function create a list of numerical values, numbers then call the function.

Step 6 : Finally, create a ‘for loop’ for getting the values as Key, Value pair.

Step 7 : Execute the program.

Program:

```
import statistics

# statistics is an inbuilt python libary

def compute_central_tendency_and_dispersion(numbers):

#Central Tendency Measures

mean = statistics.mean(numbers)

median = statistics.median(numbers)

try:

mode = statistics.mode(numbers)

except statistics.StatisticsError:

mode = "No unique mode"

# Measures of Dispersion

variance = statistics.variance(numbers)

std_deviation = statistics.stdev(numbers)

return {

"Mean": mean,

"Median": median,

"Mode": mode,

"Variance": variance,

"Standard Deviation": std_deviation
}

# You can replace “numbers” variable data with any data of your choice

numbers = [1, 2, 2, 3, 4, 4, 4, 5, 6]

results = compute_central_tendency_and_dispersion(numbers)

print("Central Tendency Measures and Measures of Dispersion:")

for key, value in results.items():

print(f"{key}: {value}")
```

OUTPUT:

Central Tendency Measures and Measures of Dispersion:

Mean: 3.4444444444444446

Median: 4

Mode: 4

Variance: 2.5277777777777777

Standard Deviation: 1.5898986690282428

RESULT:

This program has been executed successfully.

a) - Implement a Linear Regression and Multiple Linear Regression with a Real Dataset.

EX No: 02.

Date:

Aim:

Implement a Linear Regression and Multiple Linear Regression with a Real Dataset.

Algorithm:

Step 1 : Start the program.

Step 2 : Import pandas as pd, matplotlib.pyplot as plt and Then from sklearn import train_test_split, LinearRegression, mean_squared error, r2_score.

Step 3 : A CSV file named ‘ish.csv’ containing the dataset and read the dataset from the CSV file into pandas DataFrame.

Step 4 : Extract the feature column SepalLengthCm as X and extract the target column Id as y.

Step 5 : Split the dataset into training and testing subsets and set random_state to 0.

Step 6 : Create a LinearRegression model instance and fit the model using the training data (X_train, y_train).

Step 7 : Predict the target values for the test data using the trained model.

Step 8 : Compute and Calculate the values.

Step 9 : Then plot and display the values.

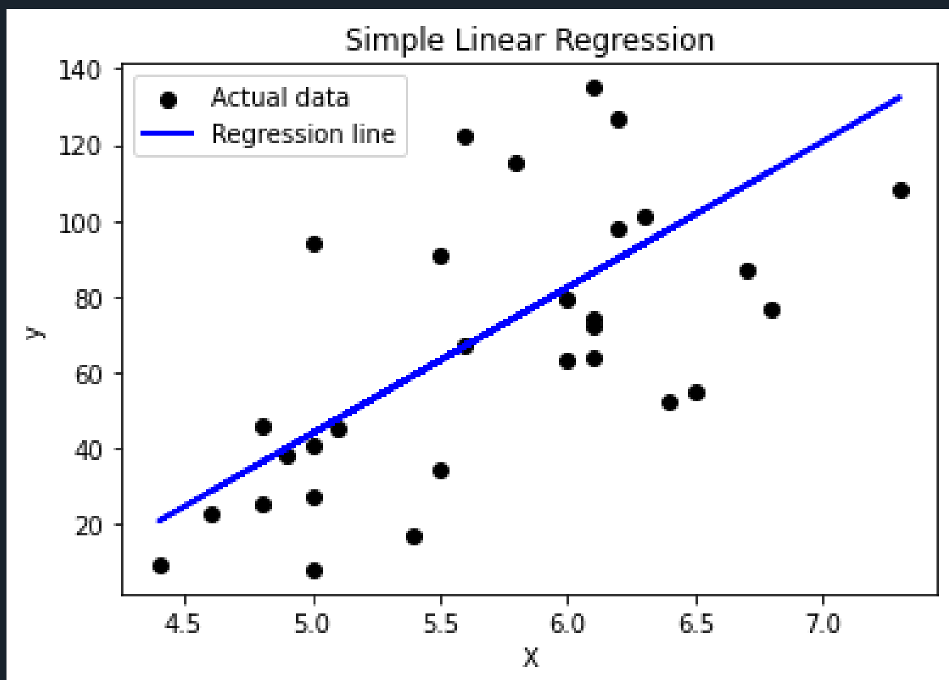
Step 10 : Execute the program.

Program:

```
import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_score

data = pd.read_csv('ish.csv')

X = data[['SepalLengthCm']]

y = data['Id']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

model = LinearRegression()

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print(f'Coefficients: {model.coef_}')

print(f'Intercept: {model.intercept_}')

# Evaluate the model

print(f'Mean squared error: {mean_squared_error(y_test, y_pred)}')

print(f'Coefficient of determination (R^2): {r2_score(y_test, y_pred)}')

# Plot the results

plt.scatter(X_test, y_test, color='black', label='Actual data')

plt.plot(X_test, y_pred, color='blue', linewidth=2, label='Regression line')

plt.title('Simple Linear Regression')

plt.xlabel('X')

plt.ylabel('y')

plt.legend()

plt.show()
```

OUTPUT:

```
Coefficients: [38.50239787]
Intercept: -148.66785148205207
Mean squared error: 810.4839226069655
Coefficient of determination (R^2): 0.3590557909341925
```



RESULT:

This program has been executed successfully.

b) - Implement a Linear Regression and Multiple Linear Regression with a Real Dataset.

EX No: 02

Date:

Aim:

Implement a Linear Regression and Multiple Linear Regression with a Real Dataset.

Algorithm:

Step 1 : Start the program.

Step 2 : Import numpy as np, pandas as pd, matplotlib.pyplot as plt and Then from sklearn import train_test_split, LinearRegression, mean_squared error, r2_score.

Step 3 : Set the random seed to 0.

Step 4 : Generate random values for two features, X1 and X2 and create the target variable y using a linear combination of X1 and X2 with added random noise.

Step 5 : Combine X1, X2, and y into a single array X, then create a DataFrame df with columns 'X1', 'X2', and 'y'.

Step 6 : Split the dataset into training and testing subsets.

Step 7 : Set random_state to 0 and store the features in X_train and X_test, and the target in y_train and y_test.

Step 8 : Create an instance of LinearRegression and fit the model using the training data (X_train, y_train).

Step 9 : Predict target values for the test data (X_test) using the trained model and store these predictions in y_pred.

Step 10 : Compute and Calculate the values.

Step 11 : Then plot and display the values.

Step 12 : Execute the program.

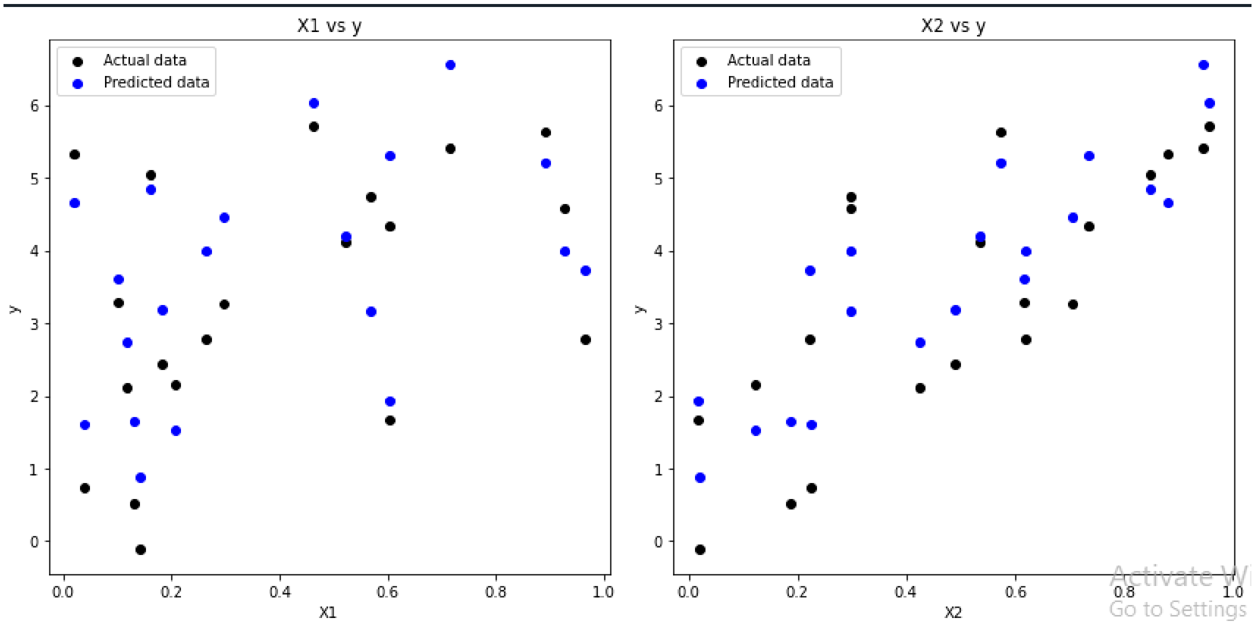
Program:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_score

np.random.seed(0)

X1 = np.random.rand(100, 1)

X2 = np.random.rand(100, 1)

y = 3 * X1 + 5 * X2 + np.random.randn(100, 1)

X = np.hstack((X1, X2))

df = pd.DataFrame(np.hstack((X, y)), columns=['X1', 'X2', 'y'])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,

random_state=0)

model = LinearRegression()

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print(f'Coefficients: {model.coef_}')

print(f'Intercept: {model.intercept_}')

print(f'Mean squared error: {mean_squared_error(y_test, y_pred)}')

print(f'Coefficient of determination (R^2): {r2_score(y_test, y_pred)}')

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
```

```
plt.scatter(X_test[:, 0], y_test, color='black', label='Actual data')

plt.scatter(X_test[:, 0], y_pred, color='blue', linewidth=1, label='Predicted data')

plt.title('X1 vs y')

plt.xlabel('X1')

plt.ylabel('y')

plt.legend()

plt.subplot(1, 2, 2)

plt.scatter(X_test[:, 1], y_test, color='black', label='Actual data')

plt.scatter(X_test[:, 1], y_pred, color='blue', linewidth=1, label='Predicted data')

plt.title('X2 vs y')

plt.xlabel('X2')

plt.ylabel('y')

plt.legend()

plt.tight_layout()

plt.show()
```

OUTPUT:

```
Coefficients: [[2.29553585 4.72302943]]
Intercept: [0.4563786]
Mean squared error: 0.7093845759110844
Coefficient of determination (R^2): 0.7668355529878372
```



RESULT:

This program has been executed successfully.

Implementation of Logistic Regression using sklearn.

EX No: 03

Date:

Aim:

Implementation of Logistic Regression using sklearn.

Algorithm:

Step 1 : Start the program.

Step 2 : Import necessary libraries such as NumPy, pandas, Matplotlib, and sklearn for data generation, manipulation, model training, and evaluation.

Step 3 : Set the random seed to 0.

Step 4 : Create random features X and a binary target y based on a threshold.

Step 5 : Store X and y in pandas DataFrame.

Step 6 : Divide the data into training (80%) and testing (20%) sets.

Step 7 : Fit a logistic regression model on the training set

Step 8 : Predict target labels on the test set.

Step 9 : Compute accuracy, confusion matrix, and classification report.

Step 10 : Calculate ROC curve and AUC.

Step 11 : Plot the ROC curve and display the AUC value.

Step 12 : Execute the program.

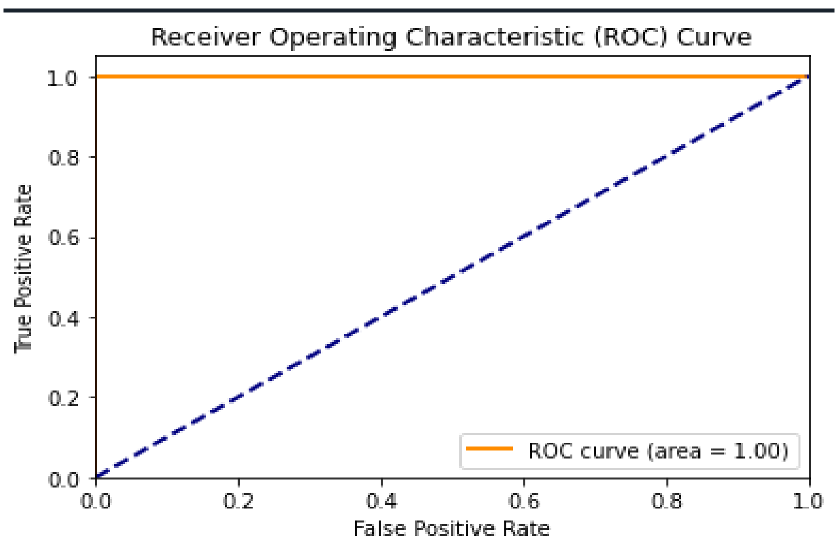
Program:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import classification_report, confusion_matrix,

accuracy_score, roc_curve, auc

np.random.seed(0)

X = np.random.rand(100, 2)

y = (X[:, 0] + X[:, 1] > 1).astype(int)

df = pd.DataFrame(np.hstack((X, y.reshape(-1, 1))), columns=['Feature1',

'Feature2', 'Target'])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,

random_state=0)

model = LogisticRegression()

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print(f'Accuracy: {accuracy_score(y_test, y_pred)}')

print('Confusion Matrix:')

print(confusion_matrix(y_test, y_pred))

print('Classification Report:')

print(classification_report(y_test, y_pred))
```

```
y_pred_proba = model.predict_proba(X_test)[:, 1]

fpr, tpr, _ = roc_curve(y_test, y_pred_proba)

roc_auc = auc(fpr, tpr)

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =
{roc_auc:.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve')

plt.legend(loc="lower right")

plt.show()
```

OUTPUT:

```
Accuracy: 1.0
Confusion Matrix:
[[ 7  0]
 [ 0 13]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         7
           1       1.00      1.00      1.00        13

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20
```



RESULT:

This program has been executed successfully.

Implement a binary classification model.

EX No: 04

Date:

Aim:

Implement a binary classification model.

Algorithm:

Step 1 : Start the program.

Step 2 : Load necessary libraries: numpy, pandas, and sklearn modules for model training and evaluation.

Step 3 : Use load_iris() to load the Iris dataset from sklearn and extract feature matrix x and target variable y.

Step 4 : Split the filtered dataset into training (80%) and testing (20%) sets using train_test_split.

Step 5 : Initialize and train a logistic regression model using the training data (x_train, y_train).

Step 6 : Use the trained model to predict class labels (y_pred) for the test data (x_test).

Step 7 : Calculate accuracy using accuracy_score.

Step 8 : Calculate and print confusion_matrix and classification_report.

Step 9 : Display the accuracy score, confusion matrix, and classification report.

Step 10 : Execute the program.

Program:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import
accuracy_score,confusion_matrix,classification_report
from sklearn.datasets import load_iris
iris=load_iris()
x=iris.data
y=iris.target
x=x[y!=2]
y=y[y!=2]
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2,random_state=4)
model=LogisticRegression()
model.fit(x_train,y_train)
y_pred=model.predict(x_test)
accuracy=accuracy_score(y_test,y_pred)
print(f'Accuracy: {accuracy:.2f}')
conf_matrix=confusion_matrix(y_test,y_pred)
print('Confusion matrix')
print(conf_matrix)
class_report=classification_report(y_test,y_pred)
```

```
print('Classification Report')

print(class_report)
```

sample dataset:

Iris.csv dataset

Id	SepalLenghCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
5	5	3.6	1.4	0.2	Iris-setosa
6	5.4	3.9	1.7	0.4	Iris-setosa
7	4.6	3.4	1.4	0.3	Iris-setosa
8	5	3.4	1.5	0.2	Iris-setosa
9	4.4	2.9	1.4	0.2	Iris-setosa
10	4.9	3.1	1.5	0.1	Iris-setosa
11	5.4	3.7	1.5	0.2	Iris-setosa
12	4.8	3.4	1.6	0.2	Iris-setosa
13	4.8	3	1.4	0.1	Iris-setosa
14	4.3	3	1.1	0.1	Iris-setosa
15	5.8	4	1.2	0.2	Iris-setosa
16	5.7	4.4	1.5	0.4	Iris-setosa
17	5.4	3.9	1.3	0.4	Iris-setosa
18	5.1	3.5	1.4	0.3	Iris-setosa
19	5.7	3.8	1.7	0.3	Iris-setosa

OUTPUT:

```
Accuracy: 1.00
Confusion matrix
[[12  0]
 [ 0  8]]
Classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        12
           1       1.00      1.00      1.00         8

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20
```

RESULT:

This program has been executed successfully.

Classification with Nearest Neighbours and NavieBayes Algorithm.

EX No: 05

Date:

Aim:

Classification with Nearest Neighbours and NavieBayes Algorithm.

Algorithm:

Step 1 : Start the program.

Step 2 : Load required libraries: datasets, train_test_split, StandardScaler, GaussianNB, KNeighborsClassifier, confusion_matrix, and classification_report from sklearn.

Step 3 : Use datasets.load_iris() to load the Iris dataset and store the feature matrix in X and target values in y.

Step 4 : Split the dataset into training and testing sets using train_test_split with 80% training data and 20% testing data (X_train, X_test, y_train, y_test) and set the random_state=1.

Step 5 : Initialize a StandardScaler object, fit and transform the training set (X_train) using StandardScaler to normalize the features and apply the same scaling to the test set (X_test).

Step 6 : Initialize a KNN classifier with n_neighbors=3, fit the KNN model on the scaled training data (X_train, y_train) and predict the class labels for the test set (X_test) using the trained KNN model.

Step 7 : Print confusion_matrix and classification_report.

Step 8 : Execute the program.

Program:

```
from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.naive_bayes import GaussianNB

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import confusion_matrix,classification_report

iris = datasets.load_iris()

X=iris.data

y=iris.target

X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.2,random_state=1)

sc=StandardScaler()

X_train=sc.fit_transform(X_train)

X_test=sc.transform(X_test)

knn=KNeighborsClassifier(n_neighbors=3)

knn.fit(X_train, y_train)

y_pred=knn.predict(X_test)

print("KNN CLASSIFICATION")

print(confusion_matrix(y_test, y_pred))

print(classification_report(y_test, y_pred))

gnb=GaussianNB()

gnb.fit(X_train,y_train)

y_pred=gnb.predict(X_test)
```

```
print("NAIVEBAYES CLASSIFICATION")

print(confusion_matrix(y_test, y_pred))

print(classification_report(y_test, y_pred))
```

OUTPUT:

```
KNN CLASSIFICATION
[[11  0  0]
 [ 0 13  0]
 [ 0  0  6]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00         6

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

NAIVEBAYES CLASSIFICATION
[[11  0  0]
 [ 0 12  1]
 [ 0  0  6]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       1.00      0.92      0.96        13
           2       0.86      1.00      0.92         6

    accuracy                           0.97        30
   macro avg       0.95      0.97      0.96        30
weighted avg       0.97      0.97      0.97        30
```

RESULT:

This program has been executed successfully.

Implementation Decision tree for classification using sklearn and its parameter tuning.

EX No: 06

Date:

Aim:

Implementation Decision tree for classification using sklearn and its parameter tuning.

Algorithm:

Step 1 : Start the program.

Step 2 : Load necessary libraries: pandas, matplotlib.pyplot, GridSearchCV, DecisionTreeClassifier, tree, confusion_matrix, classification_report, and warnings.

Step 3 : Load the Iris dataset and store features in X and target in y.

Step 4 : Use train_test_split to divide data into training (80%) and testing (20%) sets.

Step 5 : Initialize and fit a DecisionTreeClassifier on the training set and plot the decision tree structure using tree.plot_tree().

Step 6 : Predict the test set labels (y_pred) using the trained model.

Step 7 : Define hyperparameters and use GridSearchCV to find the best model.

Step 8 : Print the parameters, accuracy_score, confusion_matrix and classification_report.

Step 9 : Execute the program.

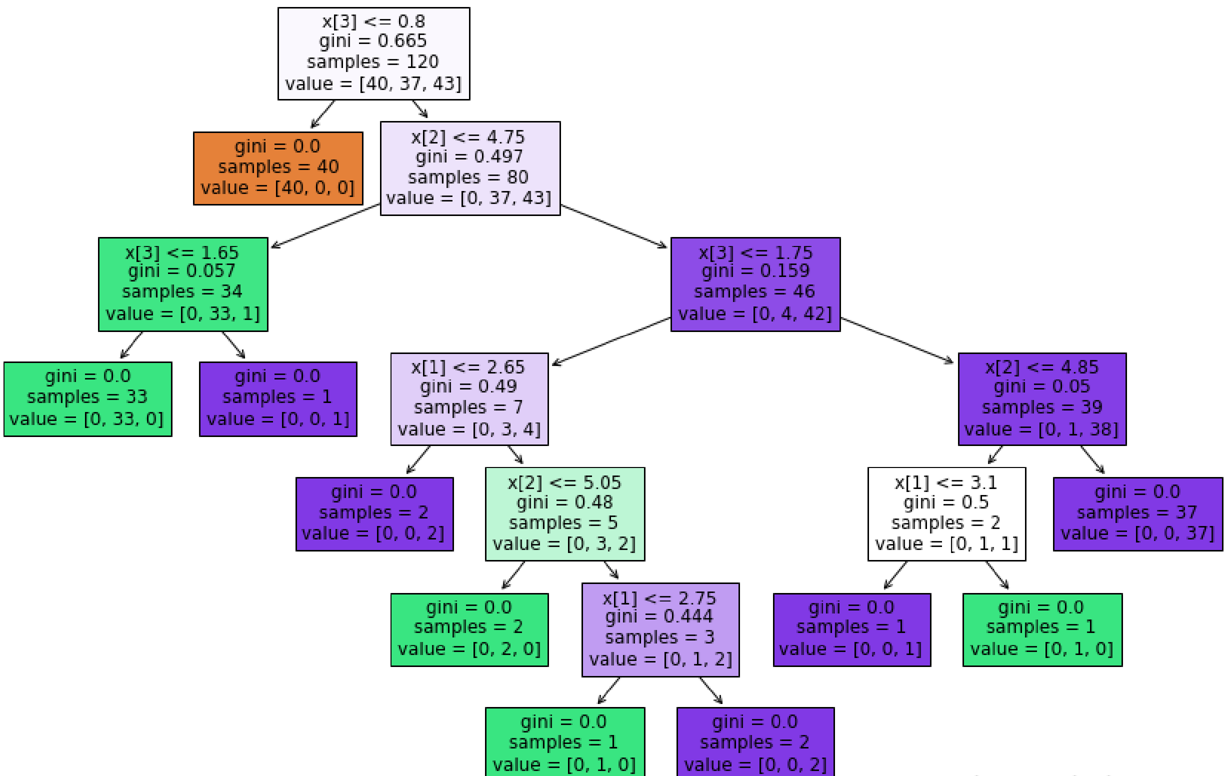
Program:

```
import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split,GridSearchCV

from sklearn.tree import DecisionTreeClassifier

from sklearn import tree

from sklearn.metrics import confusion_matrix,classification_report

import warnings

#load iris data set

iris=load_iris()

#print("train value")

#print(iris['DESCR'])

#print(iris['target'])

##independent features

X=pd.DataFrame(iris["data"],columns=['sepal length in cm','sepal width','petal
length','petal width'])

##dependent features

y=iris['target']

#print ('predicted value')

#print(iris['DESCR'])

#print(iris['target'])

##train test plot

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=0)


## apply Decision Tree Classifier
```

```
treeclassifier=DecisionTreeClassifier()

treeclassifier.fit(X_train,y_train)

DecisionTreeClassifier()

##Visualize the Decision Tree

plt.figure(figsize=(15,10))

tree.plot_tree(treeclassifier,filled=True)

y_pred=treeclassifier.predict(X_test)

#Hyperparameter

param={

    'criterion':['gini','entropy', 'log_loss'],

    'splitter':['best','random'],

    'max_depth':[1,2,3,4,5],

    'max_features':['auto','sqrt','log2']
}

treemodel=DecisionTreeClassifier()

grid=GridSearchCV(treeclassifier,param_grid=param,cv=5,scoring='accuracy')

warnings.filterwarnings('ignore')

grid.fit(X_train,y_train)

print("grid parameters",grid.best_params_)

print("grid score " , grid.best_score_)

y_pred=grid.predict(X_test)

cm=confusion_matrix(y_test,y_pred)

print("confusion matrix ",cm)

print("classification report", classification_report(y_test,y_pred))
```

OUTPUT:

```
grid parameters {'criterion': 'log_loss', 'max_depth': 2, 'max_features': 'log2', 'splitter': 'best'}
grid score  0.95
confusion matrix  [[10  0  0]
 [ 0 13  0]
 [ 0  0  7]]
classification report               precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00         7

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```



Activate Windows

RESULT:

This program has been executed successfully.

Implement the k-means algorithm.

EX No: 07

Date:

Aim:

Implement the k-means algorithm.

Algorithm:

Step 1 : Start the program.

Step 2 : Define the KMeans class with two parameters:

- n_clusters: Number of clusters (k).
-
- max_iters: Maximum iterations for convergence.

Step 3 : Randomly select initial centroids.

Step 4 : Repeat until max_iters.

Step 5 : Assign and update points to nearest centroids.

Step 6 : Create 100 random 2D points.

Step 7 : Fit the KMeans model and assign cluster labels and visualize clusters and centroids using a scatter plot.

Step 8 : Display the final clustering result.

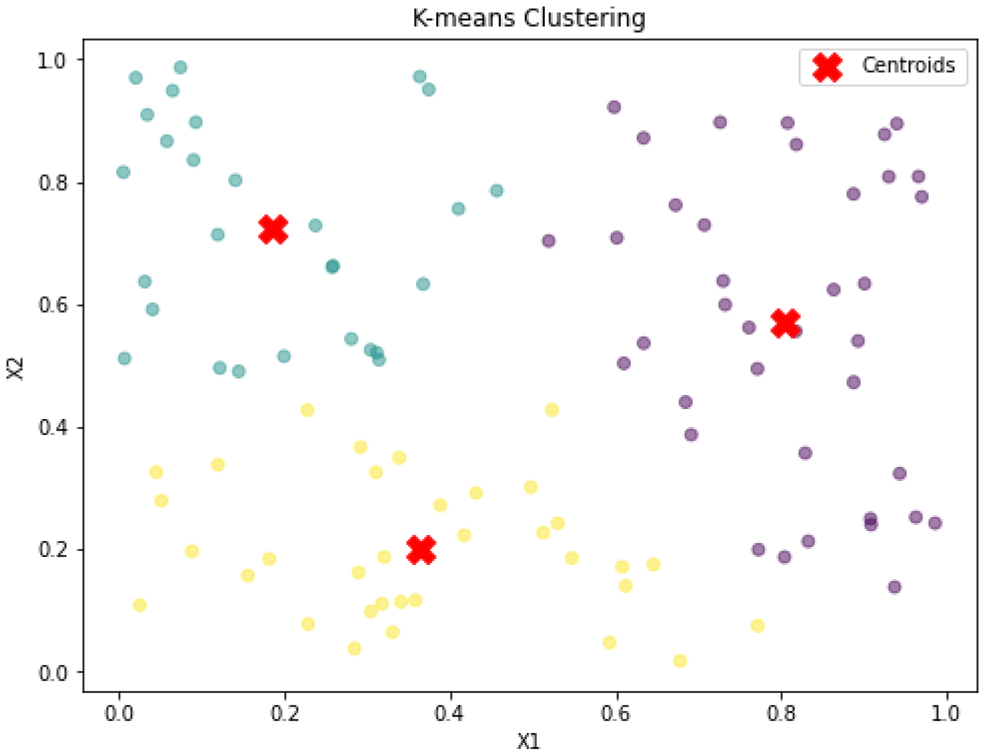
Step 9 : Execute the program.

Program:

```
import numpy as np

import matplotlib.pyplot as plt

class KMeans:

    def __init__(self, n_clusters, max_iters=100):

        self.n_clusters = n_clusters

        self.max_iters = max_iters

    def fit(self, X):

            centroids_idx = np.random.choice(X.shape[0], size=self.n_clusters,

            replace=False)

            self.centroids = X[centroids_idx]

        for _ in range(self.max_iters):

            labels = self._assign_clusters(X)

            new_centroids = np.array([X[labels == k].mean(axis=0) for k in

            range(self.n_clusters)])

            if np.allclose(self.centroids, new_centroids):

                break

            self.centroids = new_centroids

    def _assign_clusters(self, X):

        distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)

        return np.argmin(distances, axis=1)

if __name__ == "__main__":

    np.random.seed(42)

    X = np.random.rand(100, 2)
```

```
kmeans = KMeans(n_clusters=3)

kmeans.fit(X)

labels = kmeans._assign_clusters(X)

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

plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', alpha=0.5)

plt.scatter(kmeans.centroids[:, 0], kmeans.centroids[:, 1], marker='X', c='red',

 s=200, label='Centroids')

plt.title('K-means Clustering')

plt.xlabel('X1')

plt.ylabel('X2')

plt.legend()

plt.show()
```

OUTPUT:



RESULT:

This program has been executed successfully.

Implement an Image Classifier using CNN in TensorFlow/Keras.

EX No: 08

Date:

Aim:

Implement an Image Classifier using CNN in TensorFlow/Keras.

Algorithm:

Step 1 : Start the program.

Step 2 : Import necessary libraries (numpy, matplotlib, and TensorFlow/Keras modules for model building and training).

Step 3 : Load MNIST dataset, normalize pixel values and reshape data to (28, 28,1).

Step 4 : Add two Conv2D layers, each followed by MaxPooling2D and Dropout, then Flatten the data and add two Dense layers.

Step 5 : Use categorical_crossentropy loss, adam optimizer, and track accuracy

Step 6 : Fit the model for 10 epochs with validation.

Step 7 : Compute and print loss and accuracy on the test set.

Step 8 : Plot training and validation accuracy over epochs using matplotlib and Show the plot using plt.show().

Step 9 : Execute the program.

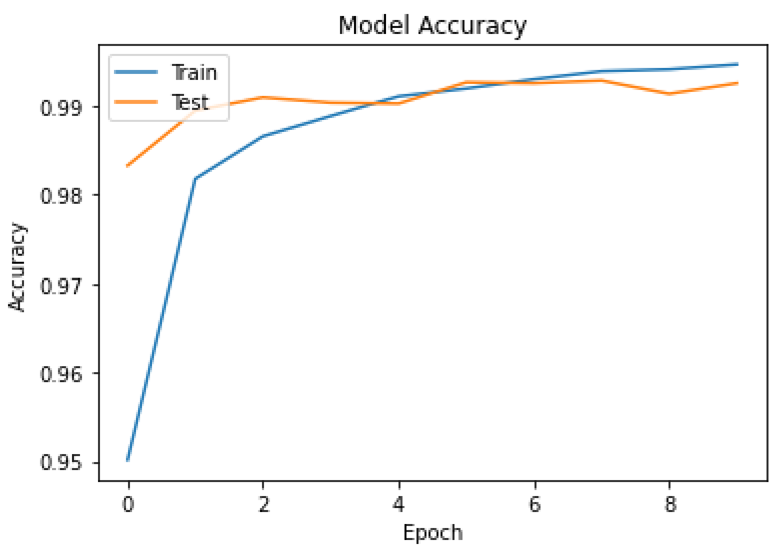
Program:

```
import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to_categorical

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
Droupout

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.astype(np.float32) / 255.0

X_test = X_test.astype(np.float32) / 255.0

X_train = np.expand_dims(X_train, axis=-1)

X_test = np.expand_dims(X_test, axis=-1)

y_train = to_categorical(y_train, num_classes=10)

y_test = to_categorical(y_test, num_classes=10)

model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28,
 28,1)))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())


model.add(Dense(128, activation='relu'))
```

```
model.add(Dense(10, activation='softmax')
model.compile(loss='categorical_crossentropy', optimizer='adam',
 metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=10, batch_size=32,
 validation_data=(X_test, y_test))
loss, accuracy = model.evaluate(X_test, y_test)
print("Loss: ", loss)
print("Accuracy: ", accuracy)
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
plt.plot(acc)
plt.plot(val_acc)
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

OUTPUT:

```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 14s 6ms/step - accuracy: 0.8882 - loss: 0.3488 - val_accuracy: 0.9833 -
val_loss: 0.0536
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9798 - loss: 0.0618 - val_accuracy: 0.9895 -
val_loss: 0.0336
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9866 - loss: 0.0426 - val_accuracy: 0.9910 -
val_loss: 0.0261
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9895 - loss: 0.0335 - val_accuracy: 0.9904 -
val_loss: 0.0283
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9913 - loss: 0.0266 - val_accuracy: 0.9903 -
val_loss: 0.0300
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9924 - loss: 0.0226 - val_accuracy: 0.9927 -
val_loss: 0.0283
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - accuracy: 0.9937 - loss: 0.0194 - val_accuracy: 0.9926 -
val_loss: 0.0211
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - accuracy: 0.9947 - loss: 0.0169 - val_accuracy: 0.9929 -
val_loss: 0.0231
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - accuracy: 0.9944 - loss: 0.0167 - val_accuracy: 0.9914 -
val_loss: 0.0305
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9951 - loss: 0.0156 - val_accuracy: 0.9926 -
val_loss: 0.0243
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9910 - loss: 0.0309
Loss:  0.024255603551864624
Accuracy:  0.9926000237464905
```



RESULT:

This program has been executed successfully.

Implement an Autoencoder in TensorFlow/Keras.

EX No: 09

Date:

Aim:

Implement an Autoencoder in TensorFlow/Keras.

Algorithm:

Step 1 : Start the program.

Step 2 : Load TensorFlow, NumPy, and Matplotlib.

Step 3 : Create input layer with shape (784,).

Step 4 : Add a dense layer with a bottleneck size of 32 and ReLU activation.

Step 5 : Add a dense layer to reconstruct the image, using sigmoid activation.

Step 6 : Load MNIST data and normalize the pixel values.

Step 7 : Fit the model using training data for 30 epochs with a batch size of 256, and validate on test data.

Step 8 : Predict reconstructed images using the autoencoder on the test set.

Step 9 : Display original and reconstructed images side by side using Matplotlib.

Step 10 : Loop through the first 10 images, showing both the original and the corresponding reconstructed image.

Step 11 : Execute the program.

Program:

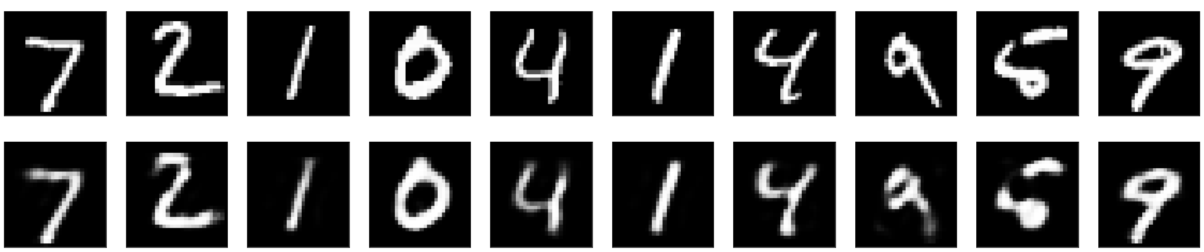
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
bottleneck=32
input_image=tf.keras.layers.Input(shape=(784,))
encoded_input=tf.keras.layers.Dense(bottleneck,activation='relu')(input_image)
decoded_output = tf.keras.layers.Dense(784, activation='sigmoid')(encoded_input)
autoencoder=tf.keras.models.Model(input_image,decoded_output)
autoencoder.compile(optimizer='adam',loss='binary_crossentropy')
(X_train,_),(X_test,_)=tf.keras.datasets.mnist.load_data()
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255
X_train =X_train.reshape((len(X_train),np.prod(X_train.shape[1:])))
X_test =X_test.reshape((len(X_test),np.prod(X_test.shape[1:])))
autoencoder.fit(X_train,X_train,epochs = 30,batch_size = 256, shuffle = True,
validation_data = (X_test,X_test))
reconstructed_img = autoencoder.predict(X_test)
n=10
plt.figure(figsize=(20,4))
for i in range(n):
    ax=plt.subplot(2,n,i+1)
    plt.imshow(X_test[i].reshape(28,28))
    plt.gray()
```

```
        ax.get_xaxis().set_visible(False)

        ax.get_yaxis().set_visible(False)

        ax=plt.subplot(2,n,i+1+n)

        plt.imshow(reconstructed_img[i].reshape(28,28))

        plt.gray()

        ax.get_xaxis().set_visible(False)

        ax.get_yaxis().set_visible(False)

    plt.show()
```

OUTPUT:

```
Epoch 1/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.3826 - val_loss: 0.1938
Epoch 2/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1829 - val_loss: 0.1545
Epoch 3/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1504 - val_loss: 0.1345
Epoch 4/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1324 - val_loss: 0.1223
Epoch 5/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1214 - val_loss: 0.1137
Epoch 6/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1137 - val_loss: 0.1079
Epoch 7/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1080 - val_loss: 0.1039
Epoch 8/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1043 - val_loss: 0.1006
Epoch 9/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1012 - val_loss: 0.0981
Epoch 10/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0988 - val_loss: 0.0963
Epoch 11/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0971 - val_loss: 0.0949
Epoch 12/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0957 - val_loss: 0.0940
Epoch 13/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0952 - val_loss: 0.0935
Epoch 14/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0947 - val_loss: 0.0932
Epoch 15/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0943 - val_loss: 0.0929
Epoch 16/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0941 - val_loss: 0.0926
Epoch 17/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0938 - val_loss: 0.0925
Epoch 18/30
```

```
Epoch 15/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0943 - val_loss: 0.0929
Epoch 16/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0941 - val_loss: 0.0926
Epoch 17/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0938 - val_loss: 0.0925
Epoch 18/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0936 - val_loss: 0.0924
Epoch 19/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0935 - val_loss: 0.0923
Epoch 20/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0935 - val_loss: 0.0923
Epoch 21/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0936 - val_loss: 0.0922
Epoch 22/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0932 - val_loss: 0.0922
Epoch 23/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0933 - val_loss: 0.0920
Epoch 24/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0933 - val_loss: 0.0921
Epoch 25/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0931 - val_loss: 0.0921
Epoch 26/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0930 - val_loss: 0.0920
Epoch 27/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0931 - val_loss: 0.0919
Epoch 28/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0930 - val_loss: 0.0919
Epoch 29/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0928 - val_loss: 0.0919
Epoch 30/30
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0930 - val_loss: 0.0919
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 687us/step
```



RESULT:

This program has been executed successfully.

Implement a Simple LSTM using TensorFlow/Keras.

EX No: 10

Date:

Aim:

Implement a Simple LSTM using TensorFlow/Keras.

Algorithm:

Step 1 : Start the program.

Step 2 : Import necessary libraries (numpy, TensorFlow/Keras modules for model building and training).

Step 3 : Create a dataset of shape (100, 10), split into X (first 9 columns) and y (last column), reshape X to 3D.

Step 4 : Create an LSTM model with 50 units, add a dense layer with 1 output.

Step 5 : Compile using adam optimizer and mse loss, train for 200 epochs.

Step 6 : Reshape test input, predict next value.

Step 7 : Print predicted value.

Step 8 : Execute the program.

Program:

```
import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM,Dense

data= np.array([[i for i in range(10)] for _ in range(100) ])

X,y=data[:,:-1],data[:,-1]

X=X.reshape((X.shape[0],X.shape[1],1))

model=Sequential()

model.add(LSTM(50,activation='relu',input_shape=(9,1)))

model.add(Dense(1))

model.compile(optimizer='adam',loss='mse')

model.fit(X,y,epochs=200,verbose=0)

test_input=np.array([7,8,9,10,11,12,13,14,15])

test_input=test_input.reshape((1,9,1))

predicted_value=model.predict(test_input,verbose=0)

print(f'Predicted value: {predicted_value[0][0]}')
```

OUTPUT:

```
In [22]: runfile('C:/Users/day1/Desktop/ish
Desktop/ishtiyaaq-aml')
Predicted value: 90.44569396972656

In [23]:
```

RESULT:

This program has been executed successfully.