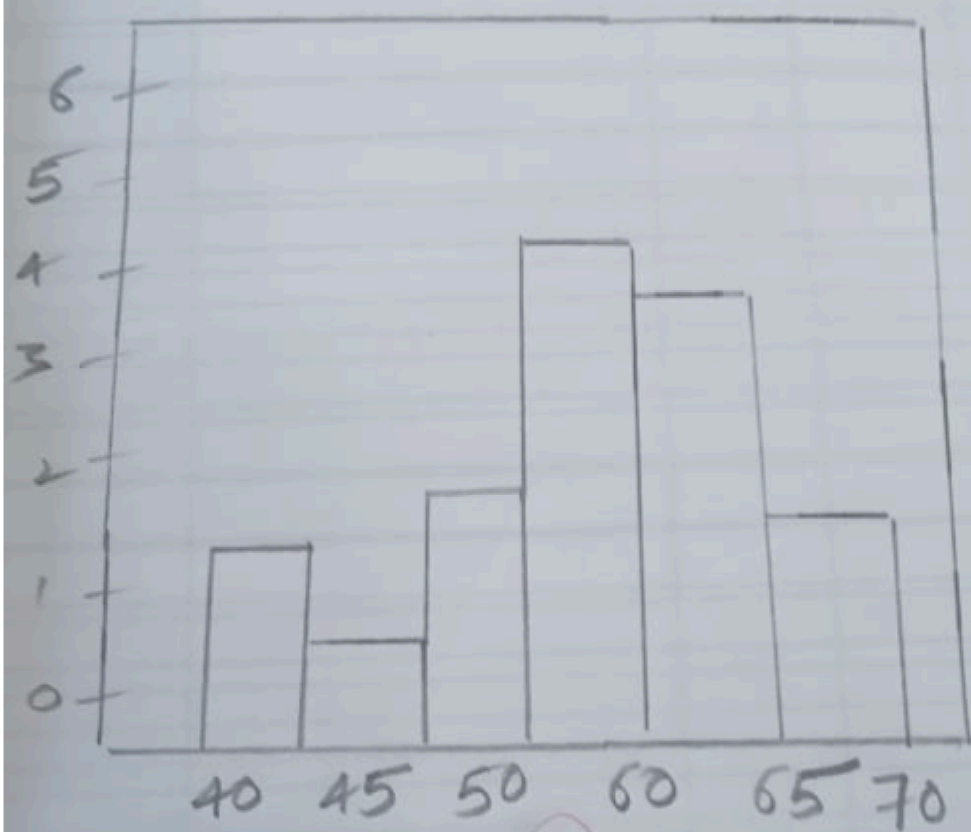
TO PERFORM TECHNIQUES FOR DATA PREPROCESSING,MEANREMOVAL,NORMALIZATION,SCALING :

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
# Load the CSV file into a DataFrame
file_path = 'your_file_path.csv'  # Replace with the actual path to your CSV
file
data = pd.read_csv(file_path, header=None, names=['Values'])
# Display the original data
print("Original Data:")
print(data)
# Mean removal
mean_values = np.mean(data['Values'])
data['Mean_Removed_Values'] = data['Values'] - mean_values
# Normalization
normalized_values = (data['Values'] - np.min(data['Values'])) /
(np.max(data['Values']) - np.min(data['Values']))
data['Normalized_Values'] = normalized_values
# Standardize the data using StandardScaler
scaler = StandardScaler()
data['Standardized_Values'] = scaler.fit_transform(data[['Values']])
# Display the preprocessed data
print("\nMean Removed Data:")
print(data[['Mean_Removed_Values']])
print("\nNormalized Data:")
print(data[['Normalized_Values']])
print("\nStandardized Data:")
print(data[['Standardized_Values']])
```

Output

	70
0	67
1	57
2	64
3	74
4	65



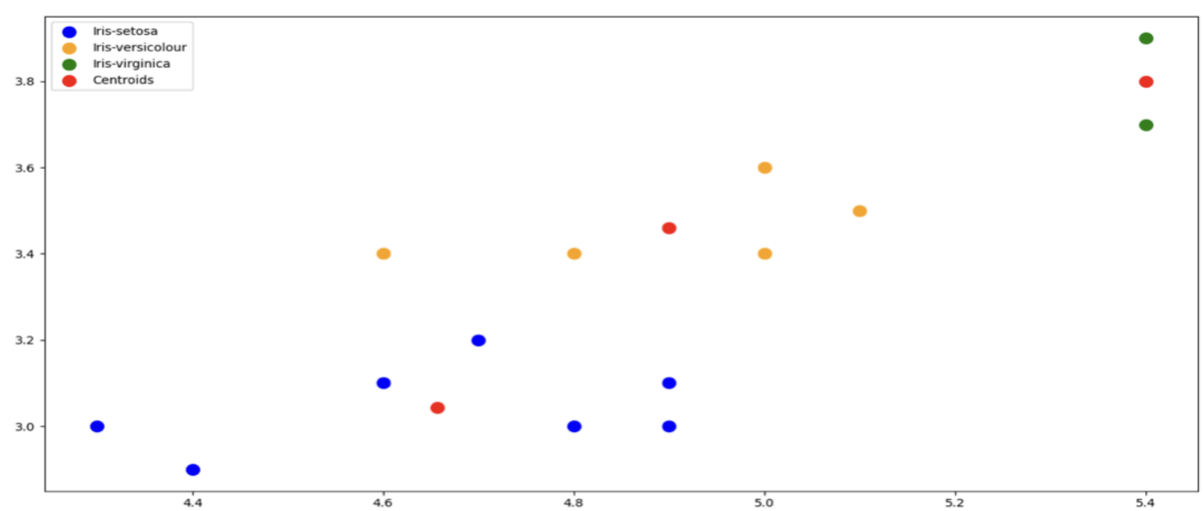
59.0

0	8.0
1	-2.0
2	5.0
3	15.0
4	6.0
5	-3.0

K-MEANS CLUSTERING :

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import MinMaxScaler
iris = pd.read_csv("Iris1.csv")
x = iris.iloc[:, [ 1,2,3,4]].values
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', max_iter = 300,
n_init = 10, random_state = 0)
    kmeans.fit(x)
    wcss.append(kmeans.inertia_)
kmeans = KMeans(n_clusters = 3, init = 'k-means++', max_iter = 300, n_init
= 10, random_state = 0)
y_kmeans = kmeans.fit_predict(x)
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s = 100, c = 'blue',
label = 'Iris-setosa')
plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s = 100, c = 'orange',
label = 'Iris-versicolour')
plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s = 100, c = 'green',
label = 'Iris-virginica')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:,1], s =
100, c = 'red', label = 'Centroids')
plt.legend()
plt.show()
```

OUTPUT :



2.CLASSIFICATION TECHNIQUES :

```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVR  # For regression instead of SVC for
classification
from sklearn.linear_model import LinearRegression  # For regression
from sklearn.tree import DecisionTreeRegressor  # For regression
from sklearn.ensemble import RandomForestRegressor  # For regression
from sklearn import datasets
from sklearn.metrics import mean_absolute_error

# Load the diabetes dataset
diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target

# Split 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)

# Naïve Bayes Regressor
nb_regressor = GaussianNB()
nb_regressor.fit(X_train, y_train)
nb_predictions = nb_regressor.predict(X_test)
nb_mae = mean_absolute_error(y_test, nb_predictions)
print(f"Naïve Bayes Regressor MAE: {nb_mae}")

# Support Vector Machine (SVM) Regressor
svm_regressor = SVR()
svm_regressor.fit(X_train, y_train)
svm_predictions = svm_regressor.predict(X_test)
svm_mae = mean_absolute_error(y_test, svm_predictions)
print(f"SVM Regressor MAE: {svm_mae}")

# Linear Regression
lr_regressor = LinearRegression()
lr_regressor.fit(X_train, y_train)
lr_predictions = lr_regressor.predict(X_test)
lr_mae = mean_absolute_error(y_test, lr_predictions)
print(f"Linear Regression MAE: {lr_mae}")

# Decision Tree Regressor
dt_regressor = DecisionTreeRegressor()
dt_regressor.fit(X_train, y_train)
```

```
dt_predictions = dt_regressor.predict(X_test)
dt_mae = mean_absolute_error(y_test, dt_predictions)
print(f"Decision Tree Regressor MAE: {dt_mae}")

# Random Forest Regressor
rf_regressor = RandomForestRegressor()
rf_regressor.fit(X_train, y_train)
rf_predictions = rf_regressor.predict(X_test)
rf_mae = mean_absolute_error(y_test, rf_predictions)
print(f"Random Forest Regressor MAE: {rf_mae}")
```

OUTPUT :

Naive Bayes classifier : 0.752

Svm classifier : 0.764

Logistic Regression : 0.786

Decision tree : 0.66

Random forest classifier : 0.730

Topic Modeling: Identifying Patterns in Text Data :

```
import csv
import re
def identify_patterns(csv_file_path, column_name):
    patterns = {}
    with open(csv_file_path, 'r') as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            text = row[column_name]
            # Example pattern: finding words that start with 'pattern'
            pattern_matches = re.findall(r'Female', text, flags=re.IGNORECASE)
            # Update patterns dictionary with matches
            for match in pattern_matches:
                if match in patterns:
                    patterns[match] += 1
                else:
                    patterns[match] = 1
    return patterns
csv_file_path = '2b Social_Network _Ads.csv'  # Update with your CSV file
path
column_name = 'Gender'     # Update with the actual column name in your
CSV file
result = identify_patterns(csv_file_path, column_name)
# Display the identified patterns and their counts
for pattern, count in result.items():
    print(f"Pattern: {pattern}, Count: {count}")
```

Output:

Pattern: Female, Count: 204

Building Bag of Words Model using NLTK :

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.probability import FreqDist
nltk.download('punkt')
nltk.download('stopwords')
def preprocess_text(text):
    stop_words = set(stopwords.words('english'))
    word_tokens = word_tokenize(text)
    filtered_words = [word.lower() for word in word_tokens if word.isalpha()
and word.lower() not in stop_words]
    return filtered_words
def create_bow_model(texts):
    all_words = []
    for text in texts:
        words = preprocess_text(text)
        all_words.extend(words)
    word_freq = FreqDist(all_words)
    bow_model = {word: freq for word, freq in word_freq.items()}
    return bow_model
# Example usage
texts = [
    "The cat sat on the mat, and the mat was comfortable.",
    "She sang a sweet song, a song that touched everyone's heart.",
    "Coding coding can be challenging, but coding is also incredibly
rewarding.",
]
bow_model = create_bow_model(texts)
# Print the Bag of Words model
print("Bag of Words Model:")
for word, freq in bow_model.items():
    print(f"{word}: {freq}")
```

Output:

Bag of Words Model:
cat: 1
sat: 1
mat: 2
comfortable: 1
sang: 1
sweet: 1
song: 2
touched: 1
everyone: 1
heart: 1
coding: 3
challenging: 1
also: 1
incredibly: 1
rewarding: 1

HIDDEN MARKOV MODEL :

```
import numpy as np
from hmmlearn import hmm
# Step 1: Define Model Parameters
n_states = 2  # Number of hidden states (Rainy and Sunny)
# Transition matrix (A): Probability of transitioning from one state to
another
trans_matrix = np.array([[0.7, 0.3], [0.4, 0.6]])
# Emission matrix (B): Probability of observing an emission given the
current state
emission_matrix = np.array([[0.1, 0.4, 0.5], [0.6, 0.3, 0.1]])
# Initial state probabilities (π): Probability distribution of starting in each
state
initial_probs = np.array([0.6, 0.4])
# Step 2: Create HMM Model
model = hmm.MultinomialHMM(n_components=n_states,
                        startprob_prior=initial_probs,
                        transmat_prior=trans_matrix,
                        n_iter=100)
# Step 3: Generate Training Data (for simplicity, you can use a pre-existing
dataset)
# Observations: 0 - Umbrella, 1 - Jacket, 2 - T-shirt
train_data = np.array([[0, 1, 2, 0, 1, 2, 0, 2, 1]])
# Reshape the array if needed
train_data = train_data.reshape(-1, 1)
# Step 4: Fit the Model
model.fit(train_data)
# Step 5: Predict States for a New Sequence
new_data = np.array([[0, 2, 1]])  # Umbrella, T-shirt, Jacket
new_data = new_data.reshape(-1, 1)
predicted_states = model.predict(new_data)
# Map numerical predictions to weather states
weather_states = ['Rainy', 'Sunny']
predicted_states_text = [weather_states[state] for state in predicted_states]
# Display Results
print("Predicted Weather States:", predicted_states_text)
```

Output:

```
Predicted Weather States: ['Rainy', 'Sunny', 'Rainy']
```

A Bot to Play Tic Tac Toe

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)
def check_winner(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or all(board[j][i] == player for
j in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player
for i in range(3)):
        return True
    return False
def is_board_full(board):
    return all(board[i][j] != " " for i in range(3) for j in range(3))
def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    players = ["X", "O"]
    current_player = players[0]
    while True:
        print_board(board)
        while True:
            row = int(input("Enter row (0, 1, or 2): "))
            col = int(input("Enter column (0, 1, or 2): "))
            if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == " ":
                break
            else:
                print("Invalid move. Try again.")
        board[row][col] = current_player
        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break
        if is_board_full(board):
            print_board(board)
            print("It's a tie!")
            break
        current_player = players[1] if current_player == players[0] else
players[0]
if __name__ == "__main__":
    tic_tac_toe()
```

OUTPUT :

```
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 1
X |   |
-----
  | O |
-----
  |   |
-----
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 2
X |   |
-----
  | O |
-----
  |   | X
-----
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 1
Invalid move. Try again.
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 2
X |   |
-----
  | O | O
-----
  |   | X
-----
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 1
Invalid move. Try again.
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 0
X |   |
-----
X | O | O
-----
  |   | X
-----
```

Concept of Heuristic Search Technique :

A Heuristic is a technique to solve a problem faster than classic methods, or to find an approximate solution when classic methods cannot. This is a kind of a shortcut as we often trade one of optimality, completeness, accuracy, or precision for speed

Heuristic Search Techniques in AI

Other names for these are Informed Search, Heuristic Search, and Heuristic Control Strategy. These are effective if applied correctly to the right types of tasks and usually demand domain-specific information.

Before move on to describe certain techniques, let's first take a look at the ones we generally observe. Below, we name a few.

- **Best-First Search**
- **A* Search**
- **Bidirectional Search**
- **Tabu Search**
- **Beam Search**
- **Simulated Annealing**
- **Hill Climbing**
- **Constraint Satisfaction Problems**

Constraint Satisfaction Problems (CSP)

Let's talk of a magic square. This is a sequence of numbers- usually integers- arranged in a square grid. The numbers in each row, each column, and each diagonal all add up to a constant which we call the *Magic Constant*. Let's implement this with Python.

A* Search Algorithm and Its Basic Concepts : A* algorithm works based on heuristic methods, and this helps achieve optimality. A* is a different form of the best-first algorithm.

When A* enters into a problem, firstly, it calculates the cost to travel to the neighboring nodes and chooses the node with the lowest cost. If The f(n) denotes the cost, A* chooses the node with the lowest f(n) value. Here 'n' denotes the neighboring nodes. The calculation of the value can be done as shown below:

f(n)=g(n)+h(n)f(n)=g(n)+h(n)

g(n) = shows the shortest path's value from the starting node to node n

h(n) = The heuristic approximation of the value of the node

The heuristic value has an important role in the efficiency of the A* algorithm. To find the best solution, you might have to use different heuristic functions according to the type of the problem. However, the creation of these functions is a difficult task, and this is the basic problem we face in AI

What is a Heuristic Function?

Essentially, a heuristic function helps algorithms to make the best decision faster and more efficiently. This ranking is based on the best available information and helps the algorithm decide the best possible branch to follow. Admissibility and consistency are the two fundamental properties of a heuristic function.

Admissibility: A heuristic function is admissible if it can effectively estimate the real distance between a node 'n' and the end node. It never overestimates; if it ever does, it will be denoted by 'd', which also denotes the accuracy of the solution.

Consistency: A heuristic function is consistent if the estimate of a given heuristic function turns out to be equal to or less than the distance between the goal (n) and a neighbor and the cost calculated to reach that neighbor.

A* is indeed a very powerful algorithm used to increase the performance of artificial intelligence. It is one of the most popular search algorithms in AI. The sky is the limit when it comes to the potential of this algorithm. However, the efficiency of an A* algorithm highly depends on the quality of its heuristic function.

Implementation with Python: In this section, we are going to find out how the A* search algorithm can be used to find the most cost-effective path in a graph. Consider the following graph below.

The numbers written on edges represent the distance between the nodes, while the numbers written on nodes represent the heuristic values. Let us find the most cost-effective path to reach from start state A to final state G using the A* Algorithm.

Let's start with node A. Since A is a starting node, therefore, the value of g(x) for A is zero, and from the graph, we get the heuristic value of A is 11, therefore

g(x) + h(x) = f(x)

0+ 11 =11

Thus for A, we can write

A=11

Now from A, we can go to point B or point E, so we compute f(x) for each of them

A → B = 2 + 6 = 8

A → E = 3 + 6 = 9

Since the cost for A → B is less, we move forward with this path and compute the f(x) for the children nodes of B

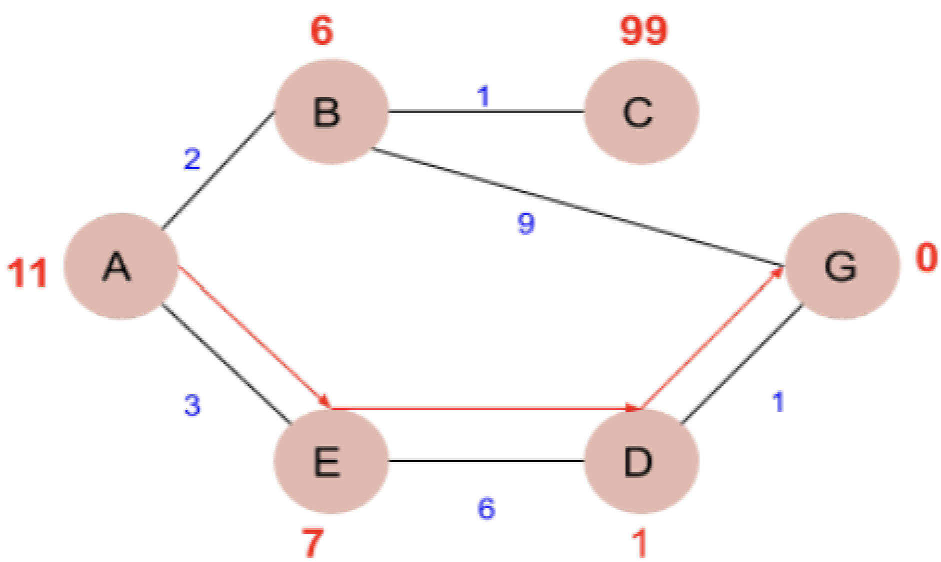
Since there is no path between C and G, the heuristic cost is set to infinity or a very high value

A → B → C = (2 + 1) + 99= 102

A → B → G = (2 + 9) + 0 = 11

Here the path A → B → G has the least cost but it is still more than the cost of A → E, thus we explore this path further
A → E → D = (3 + 6) + 1 = 10
Comparing the cost of A → E → D with all the paths we got so far and as this cost is least of all we move forward with this path. And compute the f(x) for the children of D.
A → E → D → G = (3 + 6 + 1) +0 =10
Now comparing all the paths that lead us to the goal, we conclude that A → E → D → G is the most cost-effective path to get from A to G.

OUTPUT :



SINGLE-LAYER NEURAL NETWORK :

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
import numpy as np
# Load and preprocess the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
# Split 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)
# Standardize the feature values
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Define and compile a single-layer neural network
model_single_layer = models.Sequential([
    layers.Dense(64, activation='relu', input_shape=(4,)),
    layers.Dense(3, activation='softmax')
])
model_single_layer.compile(optimizer='adam',
                           loss='sparse_categorical_crossentropy',
                           metrics=['accuracy'])
# Train the single-layer neural network
model_single_layer.fit(X_train, y_train, epochs=22, validation_data=(X_test,
y_test))
# Evaluate the single-layer model
y_pred_single_layer = np.argmax(model_single_layer.predict(X_test),
axis=1)
single_layer_accuracy = accuracy_score(y_test, y_pred_single_layer)
print(f"\nSingle-layer Neural Network - Accuracy: {single_layer_accuracy}")

# Define and compile a multi-layer neural network
model_multi_layer = models.Sequential([
    layers.Dense(64, activation='relu', input_shape=(4,)),
```

```
    layers.Dense(32, activation='relu'),
    layers.Dense(3, activation='softmax')
])
model_multi_layer.compile(optimizer='adam',
                          loss='sparse_categorical_crossentropy',
                          metrics=['accuracy'])
# Train the multi-layer neural network
model_multi_layer.fit(X_train, y_train, epochs=22, validation_data=(X_test,
y_test))
# Evaluate the multi-layer model
y_pred_multi_layer = np.argmax(model_multi_layer.predict(X_test), axis=1)
multi_layer_accuracy = accuracy_score(y_test, y_pred_multi_layer)
print(f"\nMulti-layer Neural Network - Accuracy: {multi_layer_accuracy}")T
```

>>>>>>>>OUTPUT :

Building Linear Regressor using ANN :

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(0)
X_train = np.random.rand(100, 1)
y_train = 2 * X_train + 1 + 0.1 * np.random.randn(100, 1)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=1, input_shape=(1,))
])
model.compile(optimizer='sgd', loss='mean_squared_error')
history = model.fit(X_train, y_train, epochs=100, verbose=0)
plt.plot(history.history['loss'])
plt.xlabel('Epochs')
plt.ylabel('Mean Squared Error Loss')
plt.title('Training Loss')
plt.show()
X_test = np.array([[0.2], [0.5], [0.8]])
predictions = model.predict(X_test)
for i in range(len(X_test)):
    print(f"Input: {X_test[i][0]}, Predicted Output: {predictions[i][0]}")
1/1 [==============================] - 0s 116ms/step
Input: 0.2, Predicted Output: 1.3859466314315796
Input: 0.5, Predicted Output: 2.01832914352417
Input: 0.8, Predicted Output: 2.6507115364074707
```

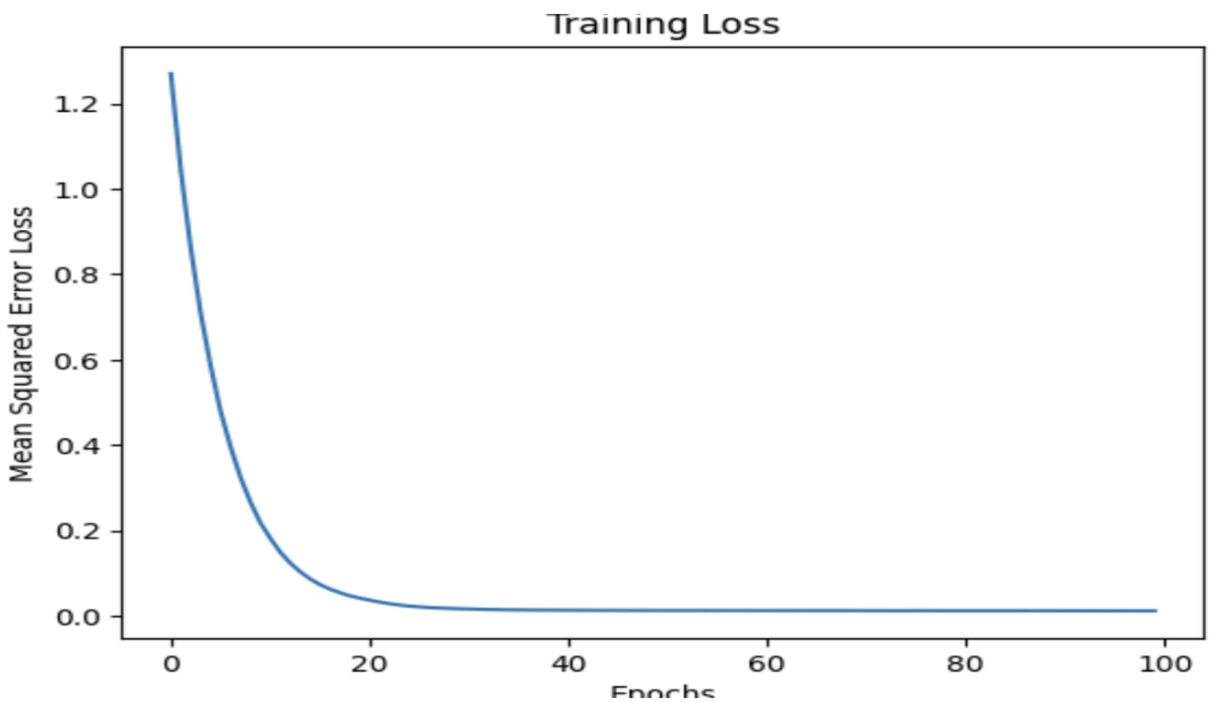


IMAGE CLASSIFIER : AN APPLICATION OF DEEP LEARNING

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, Flatten, MaxPooling2D
mnist = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train, X_test = X_train / 255.0, X_test / 255.0  # Normalize pixel values
model = Sequential([
   Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
   MaxPooling2D(pool_size=(2, 2)),
   Conv2D(64, (3, 3), activation='relu'),
   MaxPooling2D(pool_size=(2, 2)),
   Flatten(),
   Dense(64, activation='relu'),
   Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
           loss='sparse_categorical_crossentropy',
           metrics=['accuracy'])
model.fit(X_train, y_train, epochs=5, validation_split=0.1)
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'\nTest accuracy: {test_acc}')
```

OUTPUT>>>>>>>>>>>