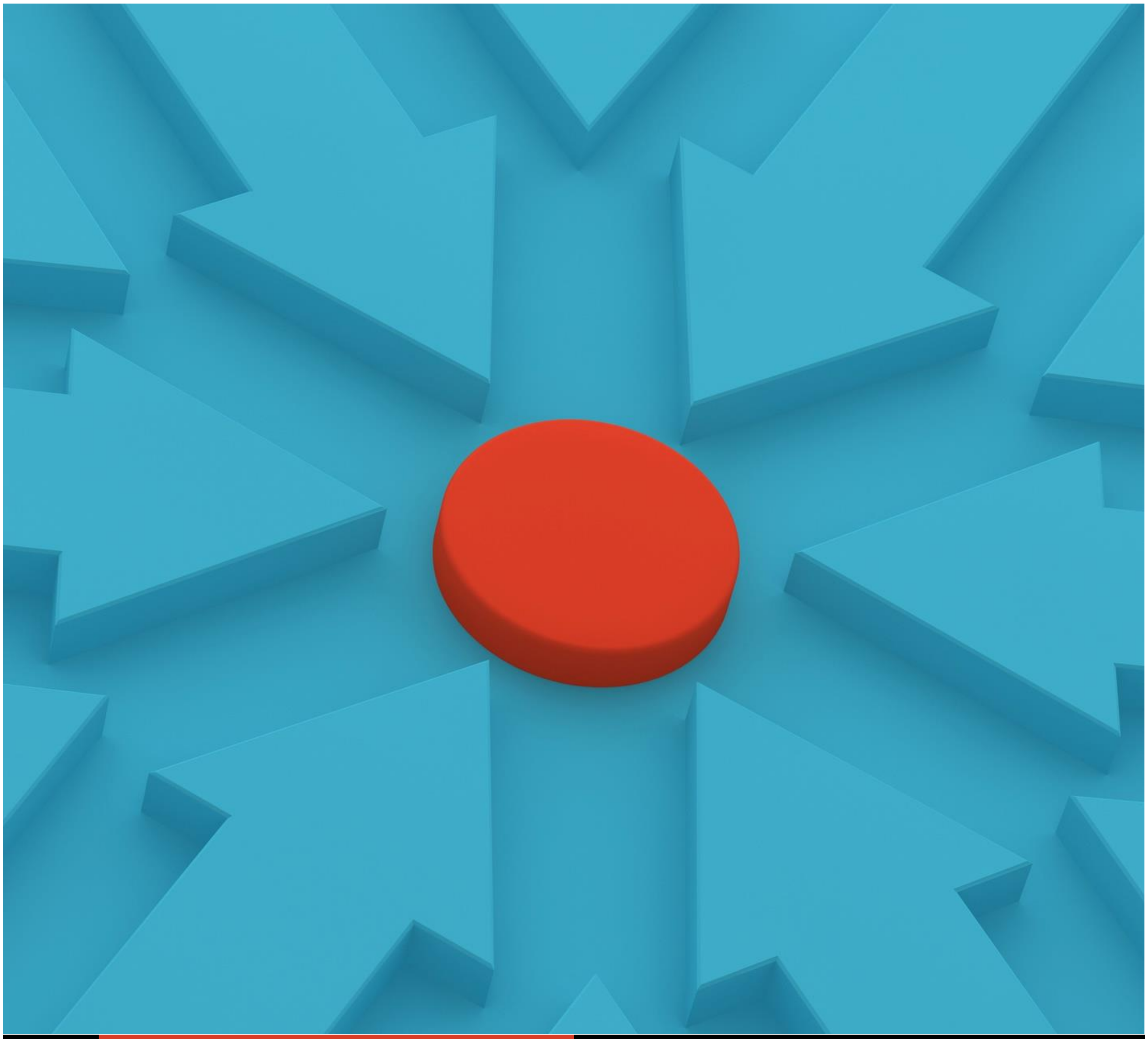
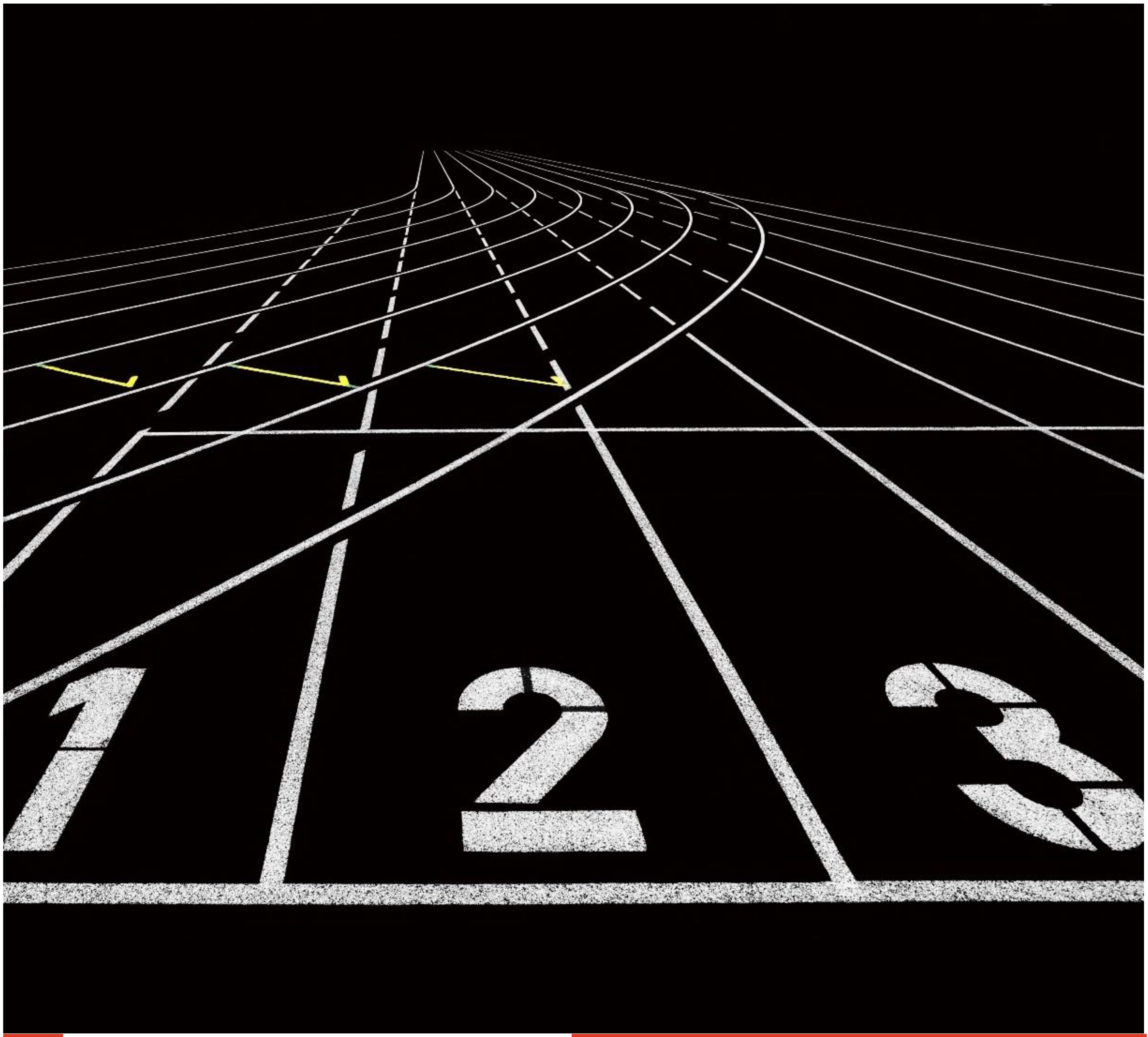


MACHINE LEARNING

ASSIGNMENT 2

LAMAN KHUDADATZADA



INTRODUCTION

THIS REPORT EXPLORES AND CREATES A LOGISTIC REGRESSION MODEL TO PREDICT WHETHER STUDENTS WILL BE ADMITTED TO A PROGRAM BASED ON THEIR EXAM SCORES. THIS MODEL HAS THE POTENTIAL TO IMPROVE THE EFFICIENCY OF THE ADMISSION PROCESS AND SUPPORT DECISION MAKING BASED ON DATA IN EDUCATIONAL CONTENT.

Importing libraries:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
```

Data Loading

Displaying the data (first few rows)

```
df = pd.read_csv('exams.csv')
print(df.head())
```

The result:

```
      exam_1     exam_2  admitted
0  34.623660  78.024693         0
1  30.286711  43.894998         0
2  35.847409  72.902198         0
3  60.182599  86.308552         1
4  79.032736  75.344376         1
```

Using min-max normalization for getting better gradien descent performance

```
def min_max_normalize(df, cols):

    scaler = MinMaxScaler()
    df[cols] = scaler.fit_transform(df[cols])
    return df
```

```
df_normalized = min_max_normalize(df, ['exam_1', 'exam_2'])
print(df_normalized.head())
```

The result:

```
     exam_1    exam_2  admitted
0  0.065428  0.694655         0
1  0.003266  0.194705         0
2  0.082968  0.619618         0
3  0.431764  0.816001         1
4  0.701943  0.655392         1
```

Visualization:

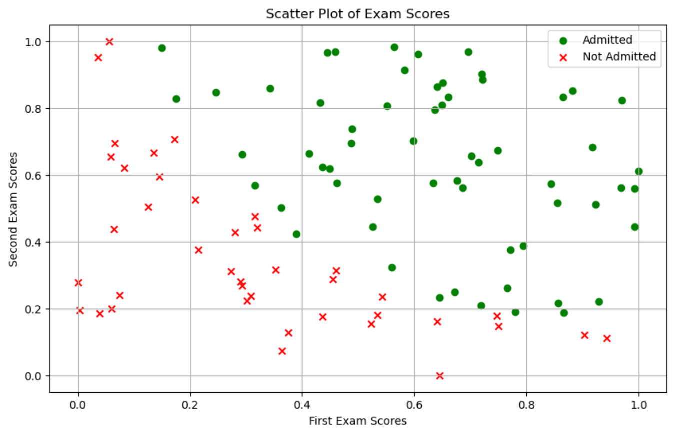
This function generates a scatter plot to visualize categorized exam scores into admitted and not-admitted

```
def plot_exam_scores(data):

    # 2 parts: admitted and not admitted
    admitted = data[data['admitted'] == 1]
    not_admitted = data[data['admitted'] == 0]
    plt.figure(figsize=(10, 6))

    # admitted students
    plt.scatter(admitted['exam_1'], admitted['exam_2'],
color='green', label='Admitted', marker='o')

    # not admitted students
    plt.scatter(not_admitted['exam_1'], not_admitted['exam_2'],
color='red', label='Not Admitted', marker='x')

    plt.xlabel('First Exam Scores')
    plt.ylabel('Second Exam Scores')
    plt.title('Scatter Plot of Exam Scores')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Calling the function:

```
plot_exam_scores(df_normalized)
```

The output:



Implementation of Logistic Regression

The x-axis in the plot shows the first exam scores of the- students, labeled as ‘First Exam Scores’. The y-axis displays the students’ scores for the second exam, labeled ‘Second Exam Scores’. The plot title is set as ‘Scatter plot of exam scores’.

The function also includes a legend to differentiate between the two groups: ‘Admitted’ and ‘Not Admitted’. It also provides a grid for easier reading. Finally, it displays the scatter plot using the following function:

plt.show()

```
x = df.drop(['admitted'], axis=1)
y = df['admitted']

#convert x and y to numpy array
x = x.to_numpy()
y = y.to_numpy()
```

```
x = np.c_[np.ones((x.shape[0], 1)), x]
y = y[:, np.newaxis]
theta = np.zeros((x.shape[1], 1))
```

a) Sigmoid of a Value

The sigmoid computes the sigmoid activation function. Sigmoid squashes output into [0, 1], interpre-ting them as odds. It takes z values or arrays.
Here are the key steps:
First, accept an input z.
Second, evaluate s=1/(1+e^(-z)) using z.
Last, return s, which is between 0 and 1.

```
def sigmoid(z):
   s = 1 / (1 + np.exp(-z))
   return s
```

b) Cost Function

The compute this cost function finds the cost for logistic regression. It uses cross-entropy loss. Evaluate model performance in training and how well it fits data.

```
def compute_cost(X, y, theta):

   m = y.size  # Number of training examples
   h = sigmoid(X @ theta)  # Predicted probabilities
(hypothesis)
   cost = -(1 / m) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))

   return cost
```

c) Gradient Descent

Gradient descent, a key function, optimizes logistic regression parameters.

```
alpha = 0.01
iterations = 100000
```

Alpha, the learning rate, controls step sizes towards the minimum cost.

```
def gradient_desc(X, y, theta):

   m = y.size
   J_history = []

   for i in range(iterations):
      h = sigmoid(X @ theta)
      error = h - y
      theta = theta - (alpha / m) * (X.T @ error)
      cost = compute_cost(X, y, theta)
      J_history.append(cost)

   return theta, J_history
```

The function iterates through specified iterations. During each iteration, parameters theta get updated. This minimizes the cost function.
The function cycles through ite-rations, methodically adjusting theta paramete-rs. Each update step brings paramete-rs closer to minimizing costs.
The end result: optimal parameter values yielding precise logistic regression model predictions.

```
wj, cost_values = gradient_desc(x, y, theta)  #cost_values =
wj_history
```

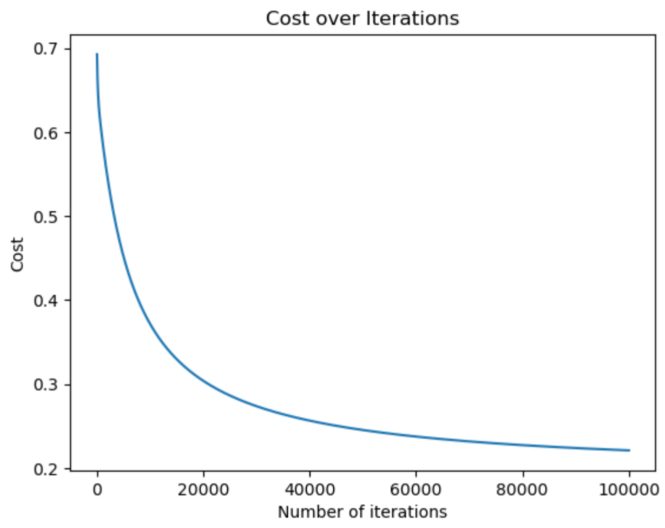
Ultimately, the goal is to find parameters that perfectly minimize costs optimizing model performance.

d) Plot the Graph of the Cost Function

The algorithm learns by adjusting parameters over time. With each iteration, it tries to lower the cost. Plotting cost over iterations shows how well the algorithm is learning. The cost should get smaller, then stop changing. That means parameters are optimized. This helps us determine whether effective learning is occurring.

```
plt.xlabel('Number of iterations')
plt.ylabel('Cost')
plt.title('Cost over Iterations')

plt.plot(cost_values)
plt.show()
```

The output:



e) Decision Boundary

This function provides a visual representation of how the logistic regression model separates the two classes (admitted and not admitted) based on the exam scores, with the decision boundary.

```
def decision_boundry(x, data, theta):

   w = np.reshape(theta, -1)

   x_values = [np.min(x[:, 1]), np.max(x[:, 2])]
   y_values = - (w[0] + np.dot(w[1], x_values)) / w[2]

   admitted = data[data['admitted'] == 1]
   not_admitted = data[data['admitted'] == 0]
```

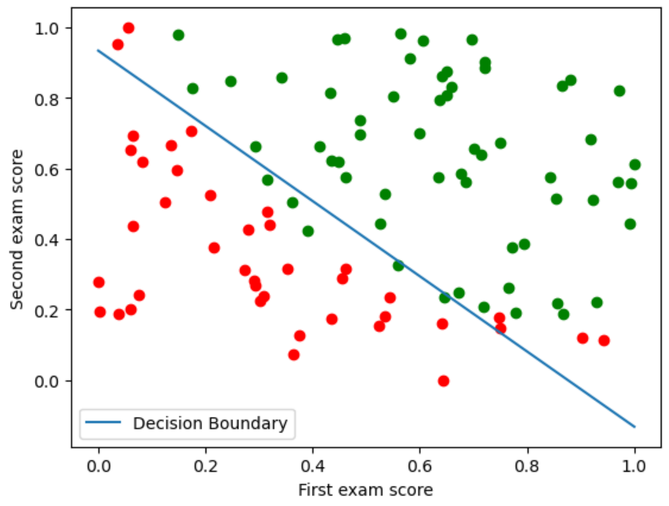
```
   plt.scatter(admitted['exam_1'], admitted['exam_2'], c='green')
   plt.scatter(not_admitted['exam_1'], not_admitted['exam_2'],
c='red')

   plt.plot(x_values, y_values, label='Decision Boundary')
   plt.xlabel('First exam score')
   plt.ylabel('Second exam score')
   plt.legend()
   plt.show()
```

Decision Boundary Computation:
The x-values for the decision boundary are set as the minimum and maximum values of the feature x[:, 1], which corresponds to the first exam score. The y-values for the decision boundary are computed using the equation of a straight line.

Calling the function:

```
decision_boundry(x, df, wj)
```

The output:



f) Predictions

The function "predict()" takes the trained logistic model. It calculates the chances of a positive outcome (admission) for each point. It then assigns labels based on the probability of meeting or exceeding 0.5. This threshold decision allows the model to sort new points. It puts each point into one of two groups (admitted or not admitted).

```
def predict(x, theta):
   probs = sigmoid(np.dot(x, theta))
   return (probs >= 0.5).astype(int)
```

The calculate_accuracy(x, y, theta) function checks how good the logistic regression model is. It first uses predict() to get predicted labels for x. Y_pred is what the model thinks the labels are. Y is the actual correct label. It compares y_pred to y. If they match, it's a 1. If not, it's a 0.
It averages these 1s and 0s. It multiplies by 100 to make it a percent.

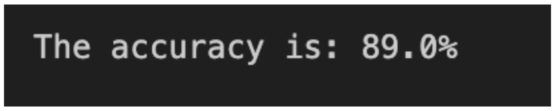
```
def calculate_accuracy(x, y, theta):
   y_pred = predict(x, theta)
   y_pred = y_pred.flatten()
   accuracy = np.mean(y_pred == y.flatten())

   return accuracy * 100
```

It shows what portion of predictions are accurate out of the whole dataset.

Displaying the accuracy of the model:

```
print(f"The accuracy is: {calculate_accuracy(x, y, wj)}%")
```

The result:



```
The accuracy is: 89.0%
```

This section of code gets the 'exam_1' and 'exam_2' features from the DataFrame. It stores these in x. The ‘admitted’ target is also extracted and stored as y. Next, a MinMaxScaler instance scales feature data x between 0 and 1.
fit_transform() applies this scaling. The scaled features are then put into df1. And y is added as a column called ‘admitted’.

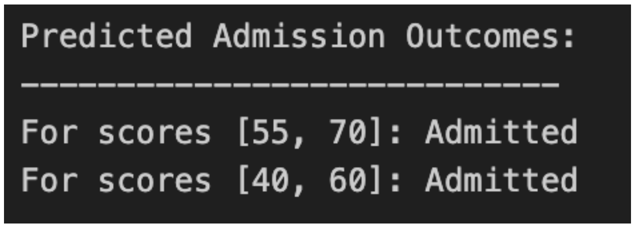
```
x = df[['exam_1', 'exam_2']].values
y = df['admitted'].values
scaler = MinMaxScaler()
x_scaled = scaler.fit_transform(x)

df1 = pd.DataFrame(x_scaled, columns=['exam_1', 'exam_2'])
df1['admitted'] = y
```

In this code, I normalized two test points using the same MinMaxScaler fitted on training data. These points represent hypothetical students’ exam scores. To predict outcomes, I added an intercept term, as logistic regression models include this. Then predict() to get admission predictions for both test points, based on learned parameters wj from the model.

```
test_1_normalized = scaler.transform([[55, 70]])
test_2_normalized = scaler.transform([[40, 60]])

# Prepare the test data points by adding the intercept term
test_1_prep = np.concatenate(([1],
test_1_normalized.flatten()))
test_2_prep = np.concatenate(([1],
test_2_normalized.flatten()))

# Make predictions using your model
result1 = predict(test_1_prep, wj)
result2 = predict(test_2_prep, wj)

print("\nPredicted Admission Outcomes:")
print("----------------------------")
print(f"For scores [55, 70]: {'Admitted' if result1 == 1 else 'Not
Admitted'}")
print(f"For scores [40, 60]: {'Admitted' if result2 == 1 else 'Not
Admitted'}")
```

Finally, printing the predicted outcomes, showing if the model predicts each student as admitted or not admitted.

The result:



```
Predicted Admission Outcomes:
----------------------------
For scores [55, 70]: Admitted
For scores [40, 60]: Admitted
```

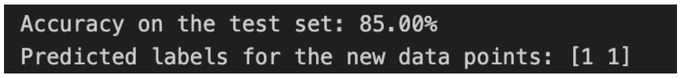
Logistic Regression Using Library

This function divides data into training and testing sets. Eighty percent becomes x_train, y_train for training. Twenty percent becomes x_test and y_test for testing.
Random_state is 42 for reproducibility.
LogisticRegression initializes a logistic regression model. Fit() trains the- model using x_train, y_train. The model learns from training data.

```
x_train, x_test, y_train, y_test = train_test_split(x_scaled, y, test_size=0.2, random_state=42)

# Initialize and train the logistic regression model
model = LogisticRegression()
model.fit(x_train, y_train)

predictions = model.predict(x_test)

accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy on the test set: {accuracy * 100:.2f}%")

new_data_points = scaler.transform([[55, 70], [40, 60]])
new_predictions = model.predict(new_data_points)

print(f"Predicted labels for the new data points: {new_predictions}")
```

The predict() method generates predictions based on the trained model. To assess the model's performance, the accuracy_score function compares the actual labels (y_test) with the predicted labels (predictions), calculating an accuracy score. This evaluates how well the model handles unseen data. Additionally, two new exam scores undergo normalization. These normalized data points are then input into the trained logistic regression model, which predicts their corresponding labels (0 or 1).

The result:



```
Accuracy on the test set: 85.00%
Predicted labels for the new data points: [1 1]
```

Conclusion:

To sum up, this report showcases the effectiveness of logistic regression in predicting student admissions based on exam results. By normalizing data, visualizing it, building a model and assessing it, we have highlighted the potential of logistic regression as a valuable technique for binary classification in educational content.