Implementation

Implementation of the Logistic Regression Algorithm

1. Consider the dataset consisting of two-dimensional and three-dimensional data points with two classes.

Explanation of the Data.py Implementation:

1. We import the necessary libraries into the project environment for preparing and preprocessing the dataset:

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

- Pandas: for reading and preparing the dataset

- Numpy: for performing calculations on the dataset

- Matplotlib: for plotting graphs

2. We import and read the dataset files into the project environment.

```python
def load_data(dataset_number=1):
    if dataset_number == 1:
        dataset = 'Datasets/Data1-blobs-2D.csv'
    elif dataset_number == 2:
        dataset = 'Datasets/Data2-circles-2D.csv'
    elif dataset_number == 3:
        dataset = 'Datasets/Data3-classification-2D.csv'
    elif dataset_number == 4:
        dataset = 'Datasets/Data4-gaussian-2D.csv'
    elif dataset_number == 5:
        dataset = 'Datasets/Data5-moon-2D.csv'
    elif dataset_number == 6:
        dataset = 'Datasets/Data6-classification-3D.csv'
    elif dataset_number == 7:
        dataset = 'Datasets/Data7-gaussian-3D.csv'
    else:
        return 0

    data = pd.read_csv(dataset)

    return data
```

The file path for each dataset is specified, and the dataset to be read by the Pandas library is determined by the dataset_number variable.

3. We perform the necessary preprocessing on the dataset:

```python
def train_test_split(dataset, split=0.2):
    msk = np.random.rand(len(dataset)) < split
    dataset_test = dataset[msk]
    dataset_train = dataset[~msk]

    return dataset_train, dataset_test


def getXY(dataset):
    y = dataset['target'].to_numpy()
    X = dataset.drop('target', axis=1).to_numpy()
    return X, y


def normalize(X_train, X_test):
    mean, std = X_train.mean(axis=0), X_train.std(axis=0)
    return (X_train - mean) / std, (X_test - mean) / std, (mean, std)
```

- The train_test_split function divides the dataset into training data (80%) and evaluation data (20%).

- The getXY function separates the data from their labels and stores the data in the X variable and the labels in the Y variable.

- The normalize function normalizes the dataset using the Z-score method.

4. We plot the dataset on a graph:

```
def visualize_data(X_train, y_train, X_test, y_test):
    if X_train.shape[1] == 2:
        # 2D plot:
        plt.scatter(X_train[y_train == 0, 0], X_train[y_train == 0, 1], color='#f39c36', alpha=0.7,
label='Train 0')
        plt.scatter(X_train[y_train == 1, 0], X_train[y_train == 1, 1], color='#1f83c2', alpha=0.7,
label='Train 1')

        plt.scatter(X_test[y_test == 0, 0], X_test[y_test == 0, 1], color='#f39c36', label='Test 0',
edgecolors='k')
        plt.scatter(X_test[y_test == 1, 0], X_test[y_test == 1, 1], color='#1f83c2', label='Test 1',
edgecolors='k')

        plt.xlabel('Feature 1')
        plt.ylabel('Feature 2')
        plt.grid()
        plt.legend()

    else:
        # 3D plot
        ax = plt.figure().add_subplot(projection='3d')
        ax.scatter(X_train[y_train == 0, 0], X_train[y_train == 0, 1], X_train[y_train == 0, 2],
color='#f39c36', alpha=0.7, label='Train 0')
        ax.scatter(X_train[y_train == 1, 0], X_train[y_train == 1, 1], X_train[y_train == 1, 2],
color='#1f83c2', alpha=0.7, label='Train 1')

        ax.scatter(X_test[y_test == 0, 0], X_test[y_test == 0, 1], X_test[y_test == 0, 2], color='#f39c36',
label='Test 0', edgecolors='k')
        ax.scatter(X_test[y_test == 1, 0], X_test[y_test == 1, 1], X_test[y_test == 1, 2], color='#1f83c2',
label='Test 1', edgecolors='k')

        ax.grid()
        ax.legend()

        return ax
```

- This function plots the training and evaluation data for each dataset on a graph.

- First, we check the dimensions of the dataset to determine whether to plot a two-dimensional or three-dimensional graph. If the number of dimensions is equal to three, a three-dimensional graph is created, and a scatter plot of the training and evaluation data is generated in the three-dimensional space. The training data is plotted in two different colors for the different classes, and the evaluation data is plotted in the same colors but with black edges.

5. We plot the decision boundary on the graph:

```python
def visualize_decision_boundary(model, max_lim=None, min_lim=None, resolution=200, ax=None):
    if min_lim is None:
        min_lim = [-1, -1]
    if max_lim is None:
        max_lim = [1, 1]

    if max_lim.shape[0] == 2:
        # 2D plot:
        x = np.linspace(min_lim[0], max_lim[0], resolution)
        y = np.linspace(min_lim[1], max_lim[1], resolution)
        x, y = np.meshgrid(x, y)
        plane = np.stack([x, y], axis=-1).reshape(-1, 2)
        h = (model.predict(plane) > 0.5).reshape(x.shape)
        plt.contour(x, y, h, 0)
    else:
        # 3D plot
        x = np.linspace(min_lim[0], max_lim[0], resolution)
        y = np.linspace(min_lim[1], max_lim[1], resolution)
        z = np.linspace(min_lim[2], max_lim[2], resolution)
        x_mesh, y_mesh, z_mesh = np.meshgrid(x, y, z)
        plane = np.stack([x_mesh, y_mesh, z_mesh], axis=-1).reshape(-1, 3)
        h = model.predict(plane).reshape(x_mesh.shape)
        z_abs = np.abs(h - 0.5)
        z_low_index = z_abs.argmin(axis=2)
        z_low = z_abs.min(axis=2)
        func = z[z_low_index]
        func[z_low > 0.01] = np.nan

        # ax.plot_surface(x_mesh[:, :, 0], y_mesh[:, :, 0], func)
        ax.scatter(x_mesh[:, :, 0], y_mesh[:, :, 0], func, color='k', marker='.')
```

- To plot the decision boundary, whether linear or nonlinear, we create a two-dimensional or three-dimensional space, depending on the dimensions of the dataset.

- We set the X, Y, and Z axes to have a resolution of 200 using the meshgrid function and combine them to create an array where each point in the two-dimensional or three-dimensional space is contained in this array.

- We then input this array into the model. The output is a series of values assigned to each point in the space by the model.

- We use the Matplotlib library and the Contour or Scatter functions to plot the points in the space that have been designated by the model as the decision boundary and have values between 0 and 1.

You can see the results of running this file in the next chapter or in the Results section .

1.1 - In order to classify this data, a logistic regression model was designed and the accuracy of the proposed method along with the final equation obtained should be reported.

1.2 - Display the decision boundary obtained for each dataset separately by this classifier. (The parameters of this algorithm should be obtained directly and using the gradient descent method.)

Explanation of the implementation of the LogisticRegression.py file, which is related to this question:

1. Necessary libraries are imported into the project environment for implementing the functions required to solve the problem.

```
import matplotlib.pyplot as plt
import numpy as np
from Data import normalize
from sklearn.preprocessing import PolynomialFeatures
```

The following libraries are imported into the project environment for implementing the required functions:

- Matplotlib: for plotting graphs
- Numpy: for performing calculations on datasets
- The Normalize function is imported from the Data.py file.

• Sklearn: for implementing non-linear logistic regression and using PolynominalFeatures.

2. Implementation of the LogisticRegression class:

```python
class LogisticRegression:
    def __init__(self, in_dim, n_iter=1000, lr=0.1):
        self.w = np.zeros((in_dim))
        self.b = np.zeros((1))
        self.n_iter = n_iter
        self.lr = lr

    def CE_loss(self, z, y):
        # h = 1 / (1 + e^-z and
        # loss = (-y * log(h) - (1 - y) * log(1 - h)).sum() / y.shape[0]
        # better approach:
        # log(h) = log(1 / (1 + e^-z))
        #        = log(1) - log(e^0 + e^-z)
        # log(1 - h) = log(e^-z / (1 + e^-z))
        #            = -z - log(e^0 + e^-z)
        loss = (y * np.logaddexp(0, -z) + (1 - y) * (z + np.logaddexp(0, -z)))
        return loss.mean()

    def fit(self, X_train, y_train, X_test, y_test):
        losses_train = []
        losses_test = []
        for i in range(self.n_iter):
            h = self.predict(X_train)
            loss_der = h - y_train
            delta_w = self.lr * np.dot(X_train.T, loss_der) / X_train.shape[0]
            delta_b = self.lr * np.sum(loss_der) / X_train.shape[0]
            self.w -= delta_w
            self.b -= delta_b

            z_train = self.get_z(X_train)
            z_test = self.get_z(X_test)
            losses_train.append(self.CE_loss(z_train, y_train))
            losses_test.append(self.CE_loss(z_test, y_test))
        return losses_train, losses_test

    def get_z(self, X):
        z = np.dot(X, self.w) + self.b
        return z

    def predict(self, X):
        z = self.get_z(X)
        return 1 / (1 + np.exp(-z))

    def score(self, X, y):
        h = self.predict(X) > 0.5
        return (h == y).mean()

    def get_equation(self):
        if len(self.w) == 2:
            return f'{self.w[0]:.4f} x0 + {self.w[1]:.4f} x1 + {self.b[0]:.4f} = 0'
        else:
            return f'{self.w[0]:.4f} x0 + {self.w[1]:.4f} x1 + {self.w[2]:.4f} x2 + {self.b[0]:.4f} = 0'
```

This class, along with the functions shown above, is used to implement logistic regression. The implementation method of the above functions is described as follows:

• The CE_loss(self, z, y) function calculates the cross-entropy loss, a measure of how well the model's predictions match the actual labels.

• The fit(self, X_train, y_train, X_test, y_test) function trains the model using the gradient descent method. According to this method, the weights and parameters of the model will be updated at each stage of learning using derivatives and gradients. Also, the Loss value is calculated at each stage of training and will be reported at the end for examining the model's results.

• The get_z(self, X) function calculates the linear combination of input features and model parameters (z = X * w + b) according to the obtained parameters.

• The predict(self, X) function calculates the model's predictions by applying the logistic function (sigmoid) to the linear combination of input features and model parameters.

• The score(self, X, y) function calculates the accuracy of the model by comparing its predictions with the actual labels.

• The get_equation(self) function returns a string representation or the final equation obtained from the model. This function also works for two-dimensional and three-dimensional input features.

Explanation of the implementation of the HomeWorkPart1.py file:

1. Necessary libraries are imported into the project environment for implementing the functions required to solve the problem.

```python
import matplotlib.pyplot as plt

from Data import load_data, train_test_split, getXY, normalize, visualize_data, visualize_decision_boundary
from LogisticRegression import LogisticRegression
```

The following libraries are imported into the project environment for implementing the required functions:

• Matplotlib: for plotting graphs

• The functions implemented in the Data.py file are imported into the project environment.

• The functions implemented in the LogisticRegression file are imported into the project environment.

2. The implemented functions are used to solve question 1.

```python
import matplotlib.pyplot as plt

from Data import load_data, train_test_split, getXY, normalize, visualize_data, visualize_decision_boundary
from LogisticRegression import LogisticRegression

for i in range(1, 8):
    dataset = load_data(i)
    dataset_train, dataset_test = train_test_split(dataset)

    X_train, y_train = getXY(dataset_train)
    X_test, y_test = getXY(dataset_test)

    X_train, X_test, (mean, std) = normalize(X_train, X_test)

    model = LogisticRegression(X_train.shape[1])
    losses_train, losses_test = model.fit(X_train, y_train, X_test, y_test)
    print(f'Linear LogisticRegression on dataset {i}, train accuracy: {model.score(X_train, y_train) * 100:.2f}, test accuracy:
{model.score(X_test, y_test) * 100:.2f}')
    print(f'Equation of decision boundary: {model.get_equation()}')

    plt.figure()
    plt.title('Training results')
    plt.plot(losses_train, 'bo-', label='Train')
    plt.plot(losses_test, 'r--', label='Test')
    plt.grid()
    plt.legend()
    plt.xlabel('Step')
    plt.ylabel('Cross Entropy')

    plt.figure()
    plt.title('Data and Decision boundary')
    ax = visualize_data(X_train, y_train, X_test, y_test)
    visualize_decision_boundary(model, X_train.max(axis=0), X_train.min(axis=0), ax=ax)

    plt.show()
```

Initially, with a loop, we read each of the datasets and go through the following steps in each stage (the implementation method of the functions used in each stage is explained in the relevant section):

1. Separate the training and evaluation data.

2. Separate the data from their labels.

3. Normalize the data.

4. Build the logistic regression model.

5. Fit the model to the data and store the error values at each stage.

6. Report the accuracy of the model and the final equation obtained by the model.

7. Plot the model's error graph relative to the training and evaluation data at each stage of training.

The results obtained in each section can be found in the Results Analysis section.

Improving Logistic Regression Algorithm:

2. Modify the logistic regression algorithm to be able to find the decision boundary in a nonlinear form. To do this, first explain your proposed method, and then, similar to the previous section:

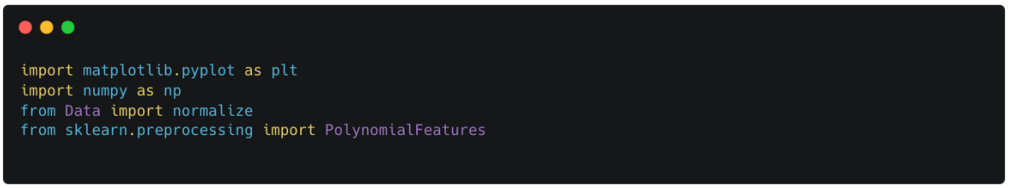
Explanation of the proposed method for solving this question:

In solving problems where there is no nonlinear decision boundary and the available data only separates with one line from each other, or in other words, the decision boundary is linear, we can separate the linear decision boundary between them by writing a linear equation, such as w1x+w2y+c=0, and entering it into the model and fitting it to the data. However, if the decision boundary between the data is not linear and we need a nonlinear decision boundary, we can produce a desired nonlinear decision boundary using a polynomial equation with a combination or multiplication of power-enabled features. For example, if a polynomial decision boundary can be constructed as $w1x^2 + w2y^2 = 0$, we will have a decision boundary in the shape of a circle or ellipse. And in this equation, if elements such as w3x or w4y are added to it, we can produce circular decision boundaries passing through different origins.

So in solving this question, using the PolynomialFeatures method, we add polynomials of a combination of power-enabled feature multiplications of the data set to the function that the model is supposed to fit to the data so that the model can produce nonlinear decision boundaries.

Explanation of the implementation of the LogisticRegression.py file and the NonLinearLogisticRegression class related to this question:

1. Import the required libraries into the project environment.



```
import matplotlib.pyplot as plt
import numpy as np
from Data import normalize
from sklearn.preprocessing import PolynomialFeatures
```

To implement the required functions in solving the question, we import the following libraries into the project environment:

• Matplotlib: for drawing graphs

• Numpy: for performing calculations on the dataset

• From the Data.py file, we import the normalize function into the project environment.

• Sklearn: to implement a nonlinear logistic regression using PolynomialFeatures.

2. Implementing nonlinear logistic regression

```python
class NonLinaerLogisticRegression(LogisticRegression):
    def __init__(self, in_dim, power=2, n_iter=1000, lr=0.1):
        self.pf = PolynomialFeatures(degree=power, include_bias=False)
        out_pf = self.pf.fit_transform(np.random.rand(1, in_dim)).shape[1]
        super().__init__(in_dim=out_pf, n_iter=n_iter, lr=lr)

    def predict(self, X):
        X = X.copy()
        if X.shape[1] != self.w.shape[0]:
            X = self.pf.transform(X)
        return super().predict(X)

    def fit(self, X_train, y_train, X_test, y_test):
        if X_train.shape[1] != self.w.shape[0]:
            X_train_new = self.pf.transform(X_train)
            X_test_new = self.pf.transform(X_test)
            return super().fit(X_train_new, y_train, X_test_new, y_test)
        else:
            return super().fit(X_train, y_train, X_test, y_test)

    def get_equation(self):
        names = self.pf.get_feature_names()
        out = ''
        for i in range(len(names)):
            if np.abs(self.w[i]) > 0.001:
                out = out + f'{self.w[i]:.4f} {names[i]} + '
        out = out + f'{self.b[0]:.4f} = 0'
        return out
```

The class related to nonlinear logistic regression inherits from the LogisticRegression class and uses its functions with the difference that:

We produce combined or polynomial features, which are power-enabled feature multiplications, to create a nonlinear decision boundary using PolynomialFeatures and enter them into the Predict, Fit, and get_equation functions. In other words, we call the mentioned functions with new inputs so that the model can fit these produced polynomials to the data to produce a nonlinear decision boundary, and its final equation is printed in the output.

2.1 Report the accuracy of the proposed method along with the final equation obtained.

2.2 Display the decision boundary obtained for each dataset separately.

(As in the previous stage, use the gradient descent-based method to obtain the network parameters.)

Explanation of the implementation of the HomeWorkPart2.py file:

1. Import the required libraries into the project environment.

```python
import matplotlib.pyplot as plt

from Data import load_data, train_test_split, getXY, normalize, visualize_data, visualize_decision_boundary
from LogisticRegression import NonLinaerLogisticRegression
```

To implement the required functions in solving the question, we import the following libraries into the project environment:

• Matplotlib: for drawing graphs

• The implemented functions in the Data.py file are imported into the project environment.

•NonLinearLogisticRegression from the LogisticRegression file is imported into the project environment.

2. Using the implemented functions to solve question two:

```python
for i in range(1, 8):
    dataset = load_data(i)
    dataset_train, dataset_test = train_test_split(dataset)

    X_train, y_train = getXY(dataset_train)
    X_test, y_test = getXY(dataset_test)

    X_train, X_test, (mean, std) = normalize(X_train, X_test)

    model = NonLinaerLogisticRegression(X_train.shape[1], power=3, lr=0.5)
    losses_train, losses_test = model.fit(X_train, y_train, X_test, y_test)
    print(f'Linear LogisticRegression on dataset {i}, train accuracy: {model.score(X_train, y_train) *
100:.2f}, test accuracy: {model.score(X_test, y_test) * 100:.2f}')
    print(f'Equation of decision boundary: {model.get_equation()}')

    plt.figure()
    plt.title('Training results')
    plt.plot(losses_train, 'bo-', label='Train')
    plt.plot(losses_train, 'r--', label='Test')
    plt.grid()
    plt.legend()
    plt.xlabel('Step')
    plt.ylabel('Cross Entropy')

    plt.figure()
    plt.title('Data and Decision boundary')
    ax = visualize_data(X_train, y_train, X_test, y_test)
    visualize_decision_boundary(model, X_train.max(axis=0), X_train.min(axis=0), ax=ax)

    plt.show()
```

First, we read each set of data in a loop and perform the following steps (the implementation method for each step is explained in the relevant section):

1. Separate the training and evaluation data.

2. Separate the data from their labels.

3. Normalize the data.

4. Build a logistic regression model with a nonlinear decision boundary.

5. Fit the model to the data and save the error values at each step.

6. Report the accuracy of the model and the final equation obtained by the model.

7. Plot the model's error rate compared to the training and evaluation data at each step of the training.

You can see the results obtained at each step in the Results section.

Obtaining optimal parameters for logistic regression using a genetic algorithm

3- In this section, you need to obtain the set of logistic regression parameters that have been improved in the previous step using a genetic algorithm. To do this, first encode these parameters. Then, using a standard genetic algorithm, search in the problem space and report the convergence plot for the average and variance of the fitness and accuracy of the best chromosome. There are no constraints on selecting the encoding type, crossover method, and mutation. The parameters of the genetic algorithm, such as crossover and mutation rates, should be adjustable.

Explanation of the implementation of the HomeWorkPart3.py file:

1- Import the required libraries into the project environment.

```
import pygad
import matplotlib.pyplot as plt
import time

from Data import load_data, train_test_split, getXY, normalize, visualize_data, visualize_decision_boundary
from LogisticRegression import NonLinaerLogisticRegression
```

To implement the required functions for solving the problem, we import the following libraries into the project environment:

- Pygad: for performing genetic algorithm calculations
- Matplotlib: for plotting graphs
- Time: for working with system time

• The implemented functions in the Data.py file are imported into the project environment.

• NonlinearLogisticRegression from the LogisticRegression file is imported into the project environment.

2- Implementation of functions related to the genetic algorithm:

```python
def decode_gene(gene):
    power = 1
    n_iter = 1
    log2lr = 0
    for i in range(3):
        power += gene[i] * (2 ** i)
    for i in range(10):
        n_iter += gene[i + 3] * (2 ** (i + 4))
    for i in range(4):
        log2lr += gene[i + 13] * (2 ** i)
    lr = 2 ** -log2lr

    return int(power), int(n_iter), lr


def fitness_function(ga_instance, solution, solution_idx):
    global X_train, y_train, X_test, y_test

    power, n_iter, lr = decode_gene(solution)
    model = NonLinaerLogisticRegression(X_train.shape[1], power=int(power), n_iter=int(n_iter), lr=lr)
    losses_train, losses_test = model.fit(X_train, y_train, X_test, y_test)

    losses_test[-1] = losses_test[-1]

    return 1 / losses_test[-1]
```

The decode_gene(gene) function: the length of a solution or chromosome in this genetic algorithm is equal to 17, and each chromosome or solution is encoded as a binary number. Bits 0 to 3 are equal to the power, which, after decoding, can ultimately be a number between 0 and 7. As mentioned in previous sections, the power is used in the input of the PolynominalFeatures function to create polynomial features with a specific power to fit the data. Then bits 3 to 13 are equal to the number of repetitions of the training process, and the last 4 bits will determine the learning rate. The learning rate will be searched

logarithmically. Therefore, the genetic algorithm tries to optimize the logistic regression model by finding the best solution to achieve the best accuracy in the output.

The fitness_function(ga_instance, solution, solution_idx) function: In the genetic algorithm, we need an objective function or fitness function for each solution or chromosome. Here, we consider the inverse of the CE error of the logistic regression model as the fitness function of the genetic algorithm. This helps us find the solution that will ultimately have the least error in the output of the logistic regression algorithm and introduce it.

3- Use of implemented functions to solve problem 3:

```
for i in range(1, 8):
    dataset = load_data(i)
    dataset_train, dataset_test = train_test_split(dataset)

    X_train, y_train = getXY(dataset_train)
    X_test, y_test = getXY(dataset_test)

    X_train, X_test, (mean, std) = normalize(X_train, X_test)

    ga_instance = pygad.GA(num_generations=5,  # Number of generations.
                           num_parents_mating=5,  # Number of parents in each generation.
                           sol_per_pop=10,  # total population size.
                           fitness_func=fitness_function,
                           num_genes=17,  # number of genes in each solution.
                           gene_space=[0, 1],
                           save_solutions=True
                           )
    ga_instance.run()
    ga_instance.plot_fitness()
    ga_instance.plot_result()
    ga_instance.plot_genes()
    ga_instance.plot_new_solution_rate()
    solution, solution_fitness, solution_idx = ga_instance.best_solution()

    power, n_iter, lr = decode_gene(solution)
    print(power, n_iter, lr)

    model = NonLinaerLogisticRegression(X_train.shape[1], power=int(power), n_iter=int(n_iter), lr=lr)
    losses_train, losses_test = model.fit(X_train, y_train, X_test, y_test)

    print(f'Linear LogisticRegression on dataset {i}, train accuracy: {model.score(X_train, y_train) * 100:.2f}, test accuracy:
{model.score(X_test, y_test) * 100:.2f}')
    print(f'Equation of decision boundary: {model.get_equation()}')

    plt.figure()
    plt.title('Training results')
    plt.plot(losses_train, 'bo-', label='Train')
    plt.plot(losses_train, 'r--', label='Test')
    plt.grid()
    plt.legend()
    plt.xlabel('Step')
    plt.ylabel('Cross Entropy')

    plt.figure()
    plt.title('Data and Decision boundary')
    ax = visualize_data(X_train, y_train, X_test, y_test)
    visualize_decision_boundary(model, X_train.max(axis=0), X_train.min(axis=0), ax=ax)

    plt.show()
```

First, with a loop, we read each of the data sets in each step and go through the following steps (the method of implementing the functions used in each step is explained in the relevant section):

1. Separate the training and evaluation data.

2. Separate the data from their labels.

3. Normalize or standardize them.

4. Run the genetic algorithm with these properties on the described chromosomes:

i. The number of generations = 5

ii. The number of parents in each generation = 5

iii. The number of solutions in each population = 10

iv. The fitness function is described.

v. The number of genes in each solution is 17, as mentioned.

5. Plot the convergence graph for the average and variance of the fitness of the population.

6. Plot the changes in each gene over time.

7. Plot the new_solution_rate over time.

8. Create a non-linear logistic regression model with the best hyperparameters found by the genetic algorithm. (Report the accuracy of the model with the best chromosome)

9. Fit the model to the data and save the error values at each step.

10. Report the accuracy of the model and the final equation obtained by the model.

11. Plot the model error graph relative to the training and evaluation data at each step of the training.

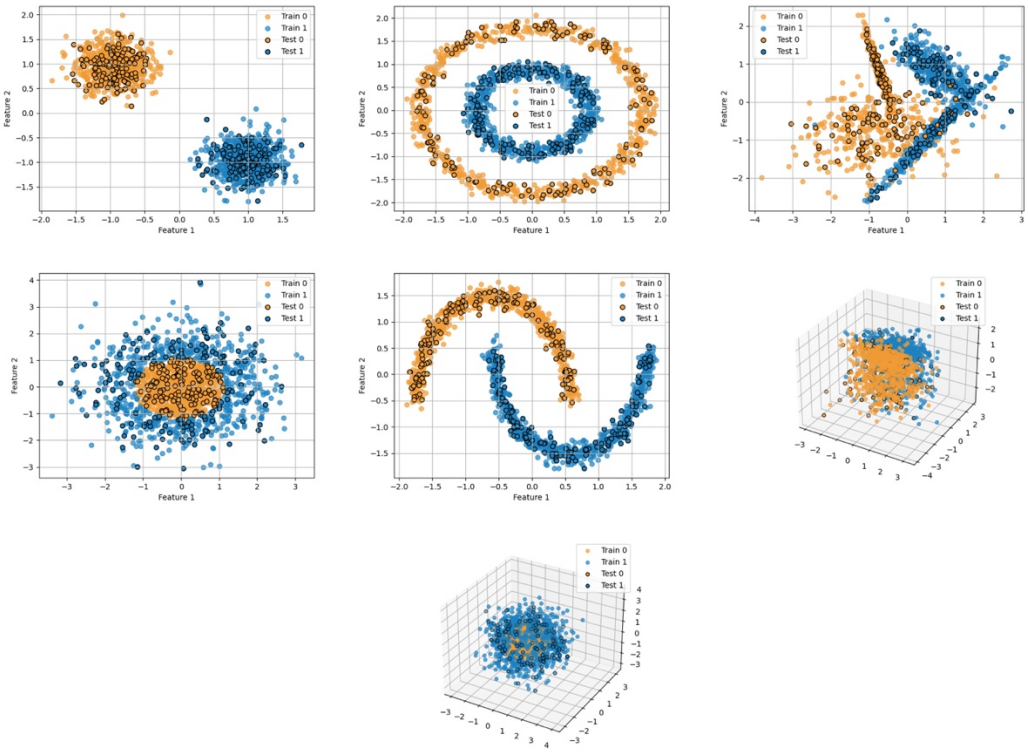
You can see the results obtained in each step in the analysis of the results.

Analysis of Results

In this report, we first reviewed the concepts of the genetic algorithm and then provided explanations related to the implemented functions. Now we move on to the results and discussion section, where we present and analyze the results of the implemented models.

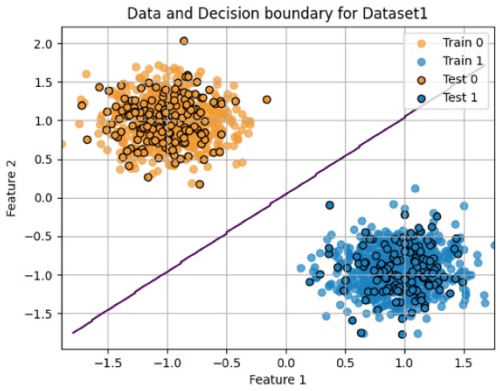
In this section, we restate the questions asked and examine the results of each part of the implementation.

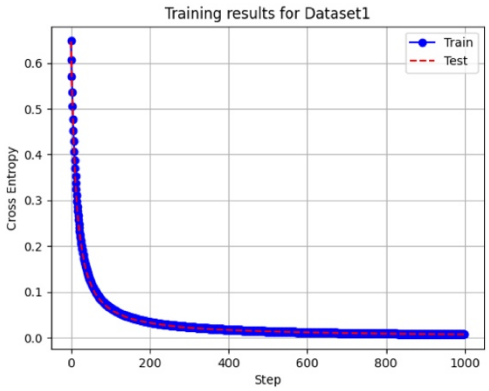
Implementation Questions Section:
Results of Data.py File Execution:
As mentioned in the implementation section, this file is responsible for reading the data set and performing the required preprocessing on them. By running this file, the data set will be plotted on the graph in order:



Question 1:
Logistic Regression Algorithm Implementation
1- Consider the dataset consisting of two-dimensional and three-dimensional data with two classes for classification.
1.1- In order to classify this data, design a logistic regression model and report the accuracy of the proposed method along with the final equation obtained.
1.2- Show the decision boundary obtained for each data set separately by this classifier.

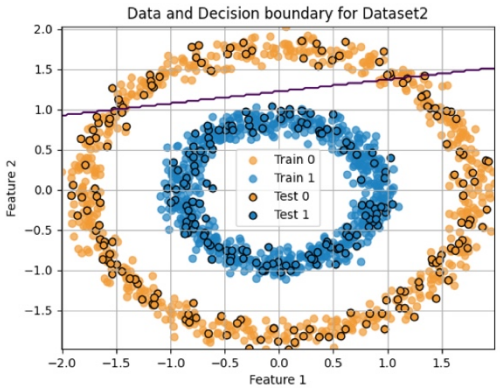
Results of HomeWorkPart1.py File Execution:
Data Set Number One:

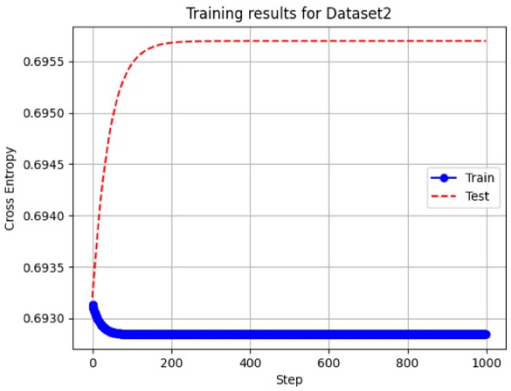





```
Linear LogisticRegression on dataset 1, train accuracy: 100.00, test accuracy: 100.00
Equation of decision boundary: 2.7952 x0 + -2.7974 x1 + 0.0884 = 0
```

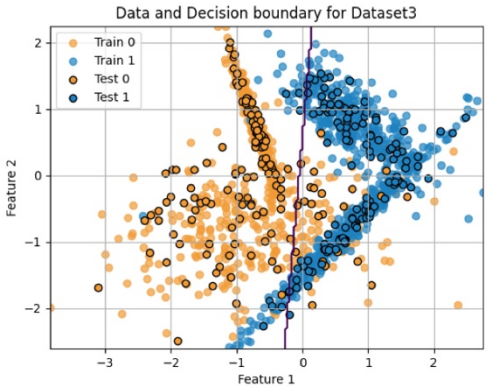
Data Set Number Two:

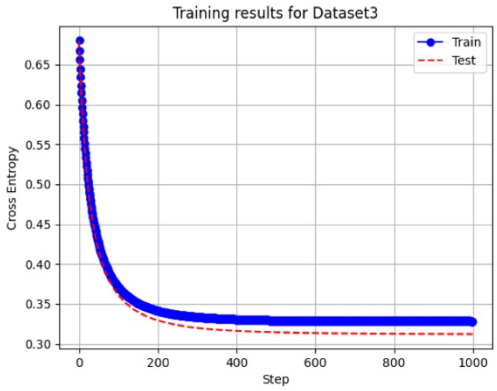





```
Linear LogisticRegression on dataset 2, train accuracy: 64.72, test accuracy: 58.54
Equation of decision boundary: 0.0046 x0 + -0.0313 x1 + 0.0379 = 0
```

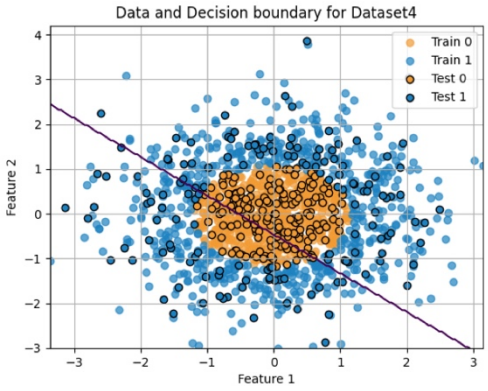
Data Set Number Three:

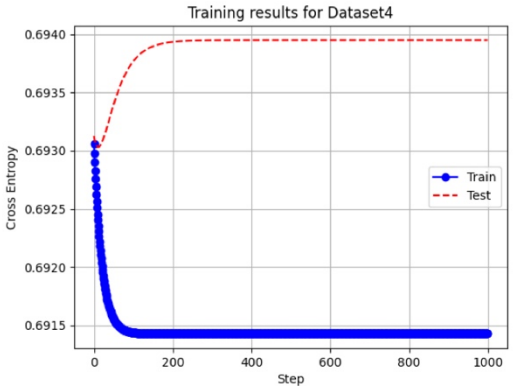





```
Linear LogisticRegression on dataset 3, train accuracy: 88.91, test accuracy: 88.69
Equation of decision boundary: 2.9058 x0 + -0.2515 x1 + 0.0936 = 0
```

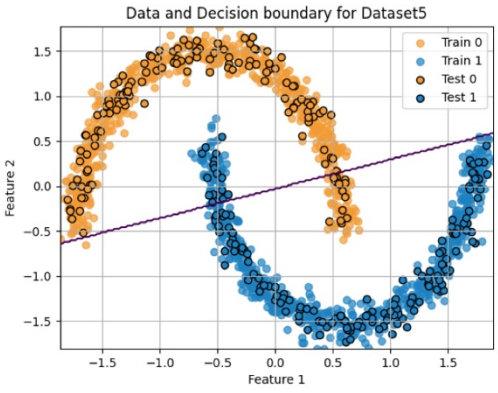
Data Set Number Four:

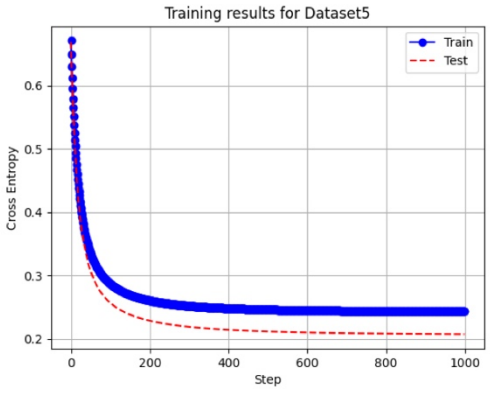





```
Linear LogisticRegression on dataset 4, train accuracy: 46.20, test accuracy: 43.67
Equation of decision boundary: 0.0723 x0 + 0.0835 x1 + 0.0373 = 0
```

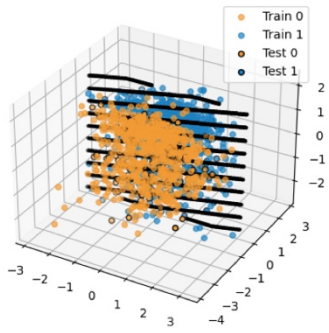
Data Set Number Five:

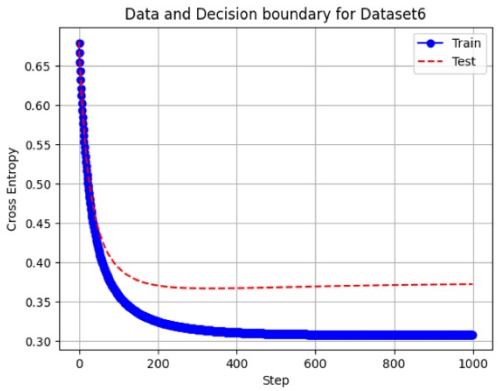





```
Linear LogisticRegression on dataset 5, train accuracy: 88.69, test accuracy: 89.90
Equation of decision boundary: 0.9711 x0 + -2.9863 x1 + -0.1156 = 0
```

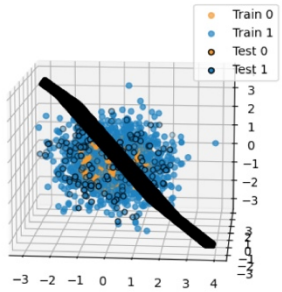
Data Set Number Six:

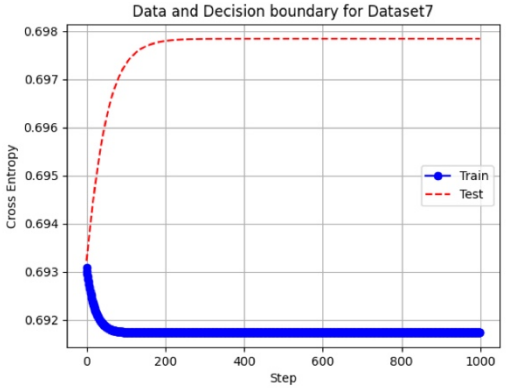





```
Linear LogisticRegression on dataset 6, train accuracy: 89.07, test accuracy: 88.37
Equation of decision boundary: 0.0398 x0 + 3.1430 x1 + -0.2021 x2 + 0.0500 = 0
```

Data Set Number Seven:



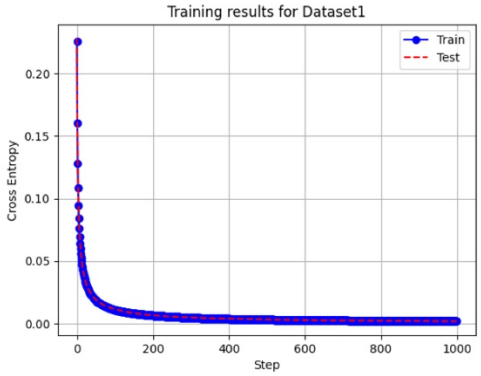



```
Linear LogisticRegression on dataset 7, train accuracy: 55.65, test accuracy: 54.01
Equation of decision boundary: 0.0832 x0 + 0.0341 x1 + 0.0557 x2 + -0.0248 = 0
```

As you can see in the above images, the model cannot achieve good accuracy on data sets that do not have a linear decision boundary. For this reason, in the next question, attempts have been made to address this issue.

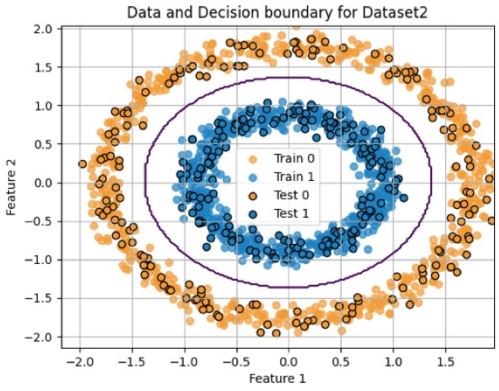
Question 2:
Improvement of Logistic Regression Algorithm
2- Modify the logistic regression algorithm to be able to find the decision boundary in a non-linear form. To do this, first explain your proposed method, and then similarly to the previous part;
2.1- Report the accuracy of the proposed method along with the final equation obtained.
2.2- Show the decision boundary obtained for each data set separately.

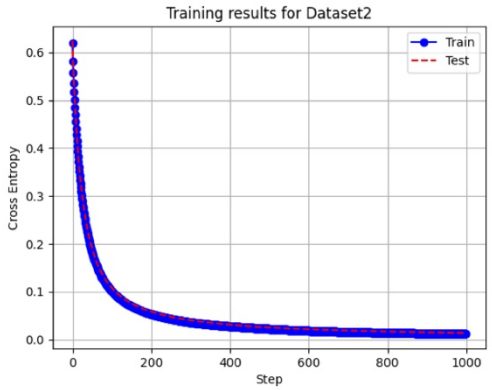
Results of HomeWorkPart2.py File Execution:
Data Set Number One:




```
Linear LogisticRegression on dataset 1, train accuracy: 100.00, test accuracy: 100.00
Equation of decision boundary: 2.3100 x0 + -2.3051 x1 + -0.0290 x0^2 + 0.0284 x0 x1 + -0.0075 x1^2 + 1.6248 x0^3 + -1.3203 x0^2 x1 + 1.3294 x0 x1^2 + -1.6339 x1^3 + -0.0318 = 0
```

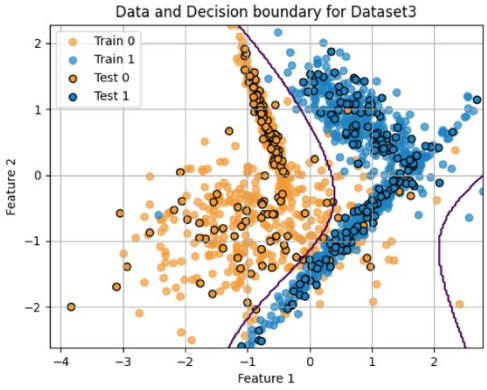
Data Set Number Two:

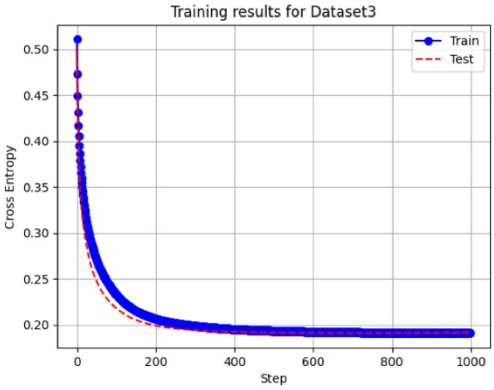





```
Linear LogisticRegression on dataset 2, train accuracy: 100.00, test accuracy: 100.00
Equation of decision boundary: -0.0010 x0 + -0.0463 x1 + -4.2308 x0^2 + -0.0700 x0 x1 + -4.2339 x1^2 + 0.0026 x0^3 + 0.0875 x0^2 x1 + 0.0541 x0 x1^2 + -0.0282 x1^3 + 7.8581 = 0
```

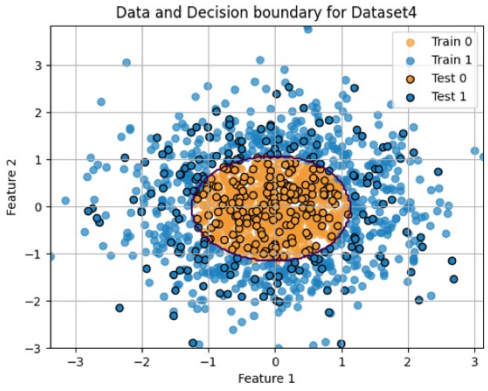
Data Set Number Three:

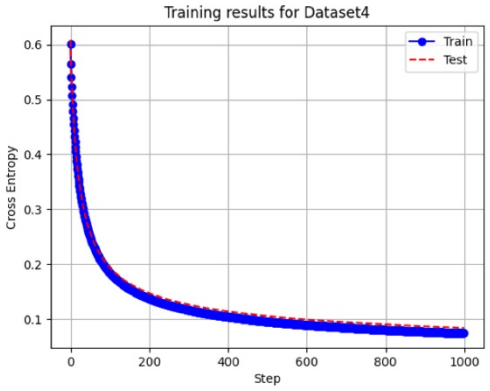





```
Linear LogisticRegression on dataset 3, train accuracy: 94.78, test accuracy: 94.88
Equation of decision boundary: 5.0706 x0 + 0.4263 x1 + -0.4769 x0^2 + 2.0450 x0 x1 + 2.0353 x1^2 + -0.3979 x0^3 + 0.3101 x0^2 x1 + 0.5387 x0 x1^2 + 0.3001 x1^3 + -1.7735 = 0
```

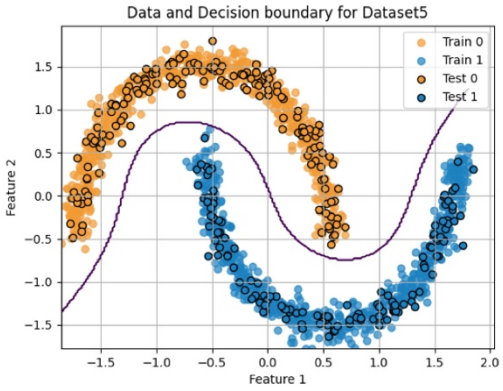
Data Set Number Four:

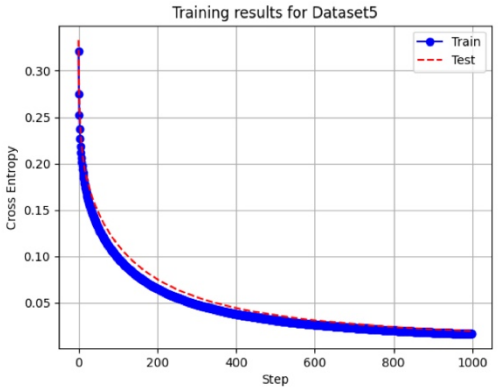





```
Linear LogisticRegression on dataset 4, train accuracy: 99.41, test accuracy: 99.05
Equation of decision boundary: 0.3732 x0 + 0.6326 x1 + 5.0035 x0^2 + -0.0173 x0 x1 + 5.6227 x1^2 + 0.1635 x0^3 + -0.1580 x0^2 x1 + -0.2539 x0 x1^2 + -0.1286 x1^3 + -7.2129 = 0
```

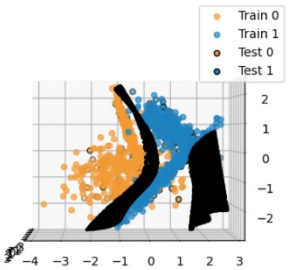
Data Set Number Five:

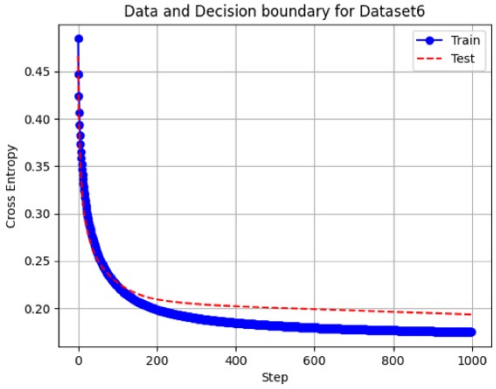





```
Linear LogisticRegression on dataset 5, train accuracy: 100.00, test accuracy: 100.00
Equation of decision boundary: -6.1262 x0 + -1.8196 x1 + -0.0767 x0^2 + -0.2937 x0 x1 + 0.1872 x1^2 + 3.6347 x0^3 + -0.5986 x0^2 x1 + -0.0659 x0 x1^2 + -2.6992 x1^3 + 0.1041 = 0
```

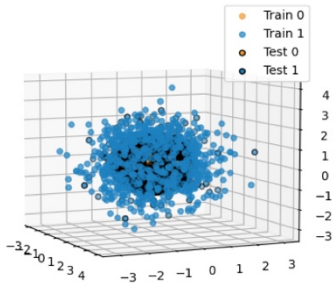
Data Set Number Six:

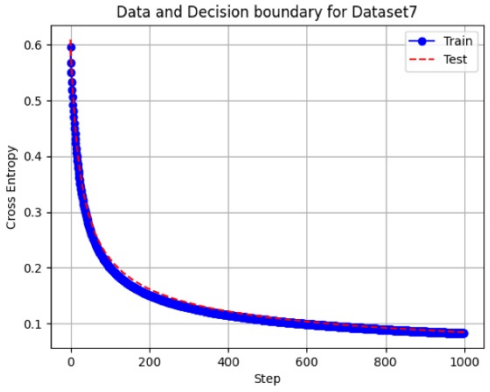





```
Linear LogisticRegression on dataset 6, train accuracy: 94.34, test accuracy: 95.02
Equation of decision boundary: 0.2206 x0 + 5.5002 x1 + 0.5482 x2 + 0.0632 x0^2 + -0.3469 x0 x1 + 0.1253 x0 x2 + -2.1429 x1^2 + 1.3681 x1 x2 + 1.8062 x2^2 + 0.0277 x0^3 + 0.0876 x0^2 x1 + -0.0218 x0^2 x2 + 0.6075 x0
```

Data Set Number Seven:






```
Linear LogisticRegression on dataset 7, train accuracy: 99.01, test accuracy: 99.66
Equation of decision boundary: 0.1293 x0 + -0.0812 x1 + 0.2806 x2 + 3.5840 x0^2 + -0.0031 x0 x1 + -0.1608 x0 x2 + 3.7725 x1^2 + 0.1312 x1 x2 + 3.5856 x2^2 + 0.0331 x0^3 + -0.0450 x0^2 x1 + -0.0822 x0^2 x2 + -0.0377
```

As you can see in the above results, the model has been able to achieve higher accuracy by producing decision boundaries with different shapes on data sets that are not linearly separable compared to the previous question. In the next question, we try to adjust the model's hyperparameters using the genetic algorithm to achieve even higher accuracy in solving this problem.

Obtaining Optimal Regression Coefficients Using Genetic Algorithm

In this section, you must obtain the set of logistic regression coefficients that were improved in the previous stage, using the genetic algorithm. To this end, first encode these parameters. Then, using the standard genetic algorithm, perform a search in the problem space and report the convergence graph for the average and variance of Fitness of the population and the accuracy of the best chromosome. There is no restriction on choosing the type of Encoding, Crossover and Mutation methods. The parameters of the genetic algorithm, the Crossover and Mutation rates, etc. should be adjustable.

Results of executing HomeWorkPart3.py file:

Graphs and results plotted for each dataset:

1. Data along with the final decision boundary

2. Error graph of the model on the training and evaluation data

3. Graph of changes in each gene over time

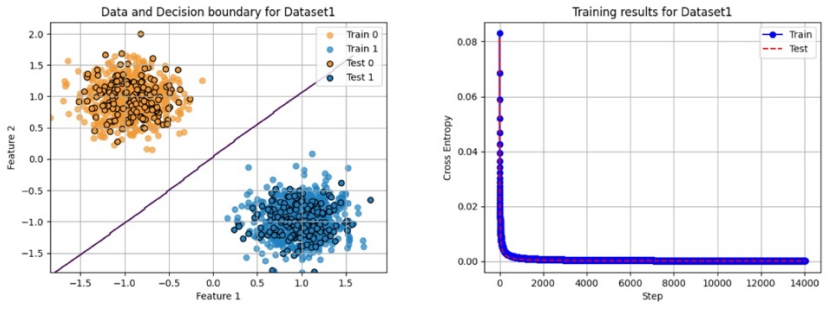
4. Convergence graph for average and variance of fitness in each generation

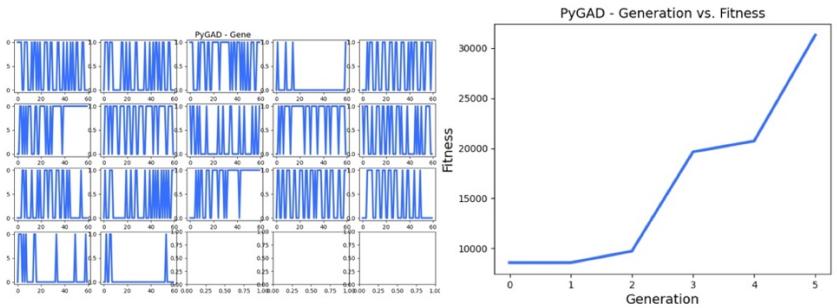
5. Solution Rate graph over time

6. Model accuracy with the best chromosome

7. Final decision boundary equation

Dataset number one:






```
Linear LogisticRegression on dataset 1, train accuracy: 100.00, test accuracy: 100.00
Equation of decision boundary: 4.3242 x0 + -4.0629 x1 + 0.0397 x0^2 + 0.1265 x0 x1 + -0.1141 x1^2 + 1.7944 x0^3 + -1.4834 x0^2 x1 + 1.5132 x0 x1^2
```

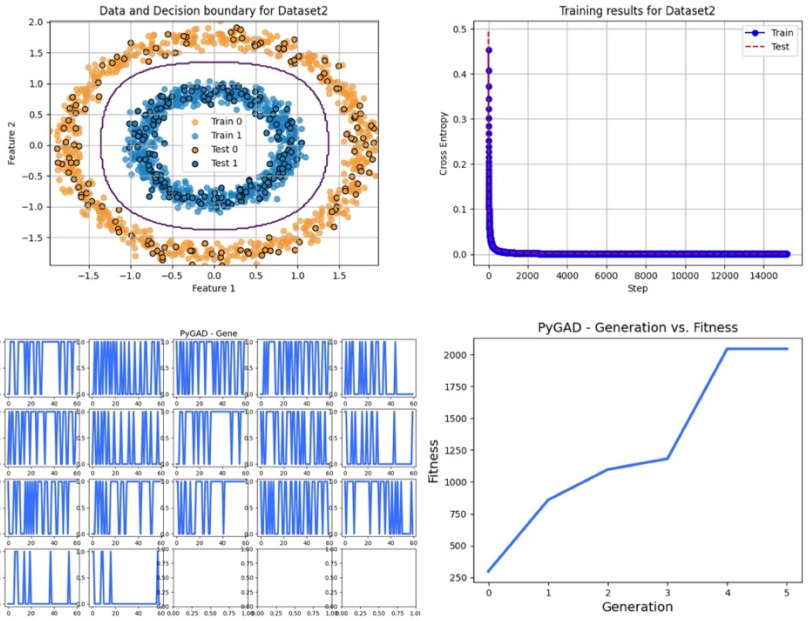
Best chromosome:

Best chromosome: power=5, n_iter=14065, lr=1.0

Final equation of decision boundary in full:

Equation of decision boundary: 4.3242 x0 + -4.0629 x1 + 0.0397 x0^2 + 0.1265 x0 x1 + -0.1141 x1^2 + 1.7944 x0^3 + -1.4834 x0^2 x1 + 1.5132 x0 x1^2 + -1.8234 x1^3 + -0.0125 x0^4 + 0.0542 x0^3 x1 + -0.0646 x0^2 x1^2 + 0.0621 x0 x1^3 + -0.0191 x1^4 + 1.3179 x0^5 + -0.9903 x0^4 x1 + 0.8858 x0^3 x1^2 + -0.9026 x0^2 x1^3 + 1.0417 x0 x1^4 + -1.3776 x1^5 + -0.0586 x0^6 + 0.0729 x0^5 x1 + -0.0703 x0^4 x1^2 + 0.0717 x0^3 x1^3 + -0.0755 x0^2 x1^4 + 0.0783 x0 x1^5 + -0.0595 x1^6 + 1.5403 x0^7 + -1.0897 x0^6 x1 + 0.8934 x0^5 x1^2 + -0.8192 x0^4 x1^3 + 0.8339 x0^3 x1^4 + -0.9403 x0^2 x1^5 + 1.1791 x0 x1^6 + -1.6656 x1^7 + -0.1786 x0^8 + 0.1608 x0^7 x1 + -0.1359 x0^6 x1^2 + 0.1250 x0^5 x1^3 + -0.1240 x0^4 x1^4 + 0.1326 x0^3 x1^5 + -0.1529 x0^2 x1^6 + 0.1876 x0 x1^7 + -0.2287 x1^8 + 0.5976 = 0

Dataset number two:




```
Linear LogisticRegression on dataset 2, train accuracy: 100.00, test accuracy: 100.00
Equation of decision boundary: 0.0065 x0 + -0.0676 x1 + 3.3221 x0^2 + -0.0133 x0 x1 + 3.2205 x1^2 + 0.1171 x0^3 + -0.0395 x0^2 x1 +
```

Best chromosome:

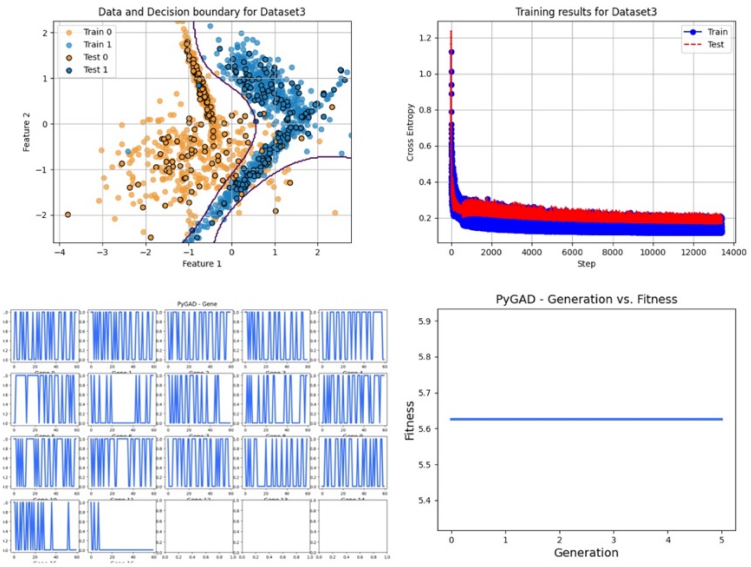
Best chromosome: power=8, n_iter=14241, lr=1.0

Final equation of decision boundary in full:

Equation of decision boundary: 0.0065 x0 + -0.0676 x1 + 3.3221 x0^2 + -0.0133 x0 x1 + 3.2205 x1^2 + 0.1171 x0^3 + -0.0395 x0^2 x1 + -0.0653 x0 x1^2 + -0.1188 x1^3 + 2.2737 x0^4 + -0.0865 x0^3 x1 + 0.0895 x0^2 x1^2 + -0.0094 x0 x1^3 + 2.0229 x1^4 + 0.1302 x0^5 + -0.0441 x0^4 x1 + 0.0566 x0^3 x1^2 + 0.0774 x0^2 x1^3 + -0.0569 x0 x1^4 + -0.2158 x1^5 + 0.5072 x0^6 + -0.1330 x0^5 x1 + -0.3673 x0^4 x1^2 + -0.0833 x0^3 x1^3 + -0.3746 x0^2 x1^4 + -0.0024 x0 x1^5 + 0.1577 x1^6 + 0.0465 x0^7 + -0.0592 x0^6 x1 + 0.0921 x0^5 x1^2 + 0.1081 x0^4 x1^3 + 0.0770 x0^3 x1^4 + 0.1020 x0^2 x1^5 + -0.0697 x0 x1^6 + -0.2116 x1^7 + -4.3764 x0^8 + -0.1978 x0^7 x1 + -0.9975 x0^6 x1^2 + -0.1097 x0^5 x1^3 + -0.6770 x0^4

x1^4 + -0.0878 x0^3 x1^5 + -0.9850 x0^2 x1^6 + 0.0025 x0 x1^7 + -4.6595 x1^8 + 7.5893 = 0

Dataset number three:




```
Linear LogisticRegression on dataset 3, train accuracy: 96.05, test accuracy: 94.72
Equation of decision boundary: 5.1999 x0 + 0.6163 x1 + -0.0701 x0^2 + -0.3817 x0 x1 + 1.6153 x1^2 + -0.9375 x0^3 +
```

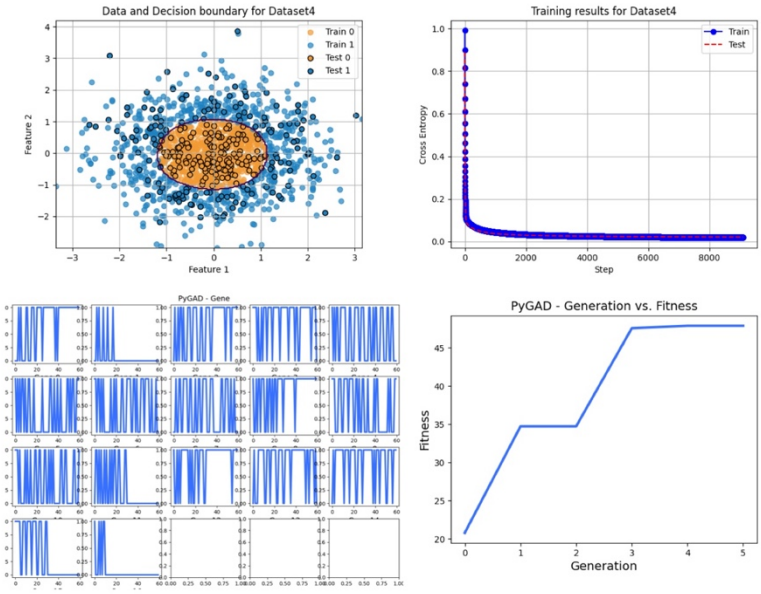
Best chromosome:

Best chromosome: power=2, n_iter=9457, lr=0.25

Final equation of decision boundary in full:

Equation of decision boundary: 5.1999 x0 + 0.6163 x1 + -0.0701 x0^2 + -0.3817 x0 x1 + 1.6153 x1^2 + -0.9375 x0^3 + -0.5045 x0^2 x1 + 2.5153 x0 x1^2 + -0.3306 x1^3 + 0.1481 x0^4 + 1.2609 x0^3 x1 + -0.6206 x0^2 x1^2 + -0.3439 x0 x1^3 + 2.1174 x1^4 + 0.1050 x0^5 + -0.3521 x0^4 x1 + 0.2572 x0^3 x1^2 + 1.2839 x0^2 x1^3 + 0.3941 x0 x1^4 + 1.3685 x1^5 + -0.0225 x0^6 + -0.0780 x0^5 x1 + -0.1768 x0^4 x1^2 + 0.7231 x0^3 x1^3 + -1.9516 x0^2 x1^4 + 1.6425 x0 x1^5 + 0.0358 x1^6 + -2.8197 = 0

Dataset number four:




```
Linear LogisticRegression on dataset 4, train accuracy: 99.83, test accuracy: 99.38
Equation of decision boundary: 0.3286 x0 + -0.1175 x1 + 1.7025 x0^2 + -0.1658 x0 x1 + -4.0376 x1^2 + 0.5947 x0^3 +
```

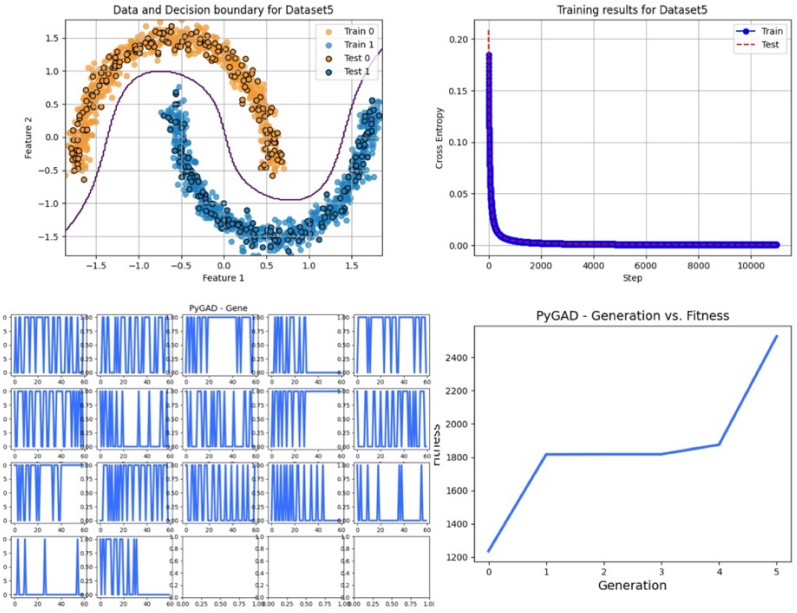
Best chromosome:

Best chromosome: power=8, n_iter=11425, lr=1.0

Final equation of decision boundary in full:

Equation of decision boundary: 0.3286 x0 + -0.1175 x1 + 1.7025 x0^2 + -0.1658 x0 x1 + -4.0376 x1^2 + 0.5947 x0^3 + 1.5912 x0^2 x1 + -0.0453 x0 x1^2 + -1.2507 x1^3 + -3.2731 x0^4 + 0.6467 x0^3 x1 + 10.7575 x0^2 x1^2 + -0.6417 x0 x1^3 + -8.3860 x1^4 + 0.6564 x0^5 + 1.0474 x0^4 x1 + -0.1906 x0^3 x1^2 + -0.0923 x0^2 x1^3 + 0.1451 x0 x1^4 + -0.5046 x1^5 + -5.4651 x0^6 + 0.1835 x0^5 x1 + 8.3439 x0^4 x1^2 + 0.6599 x0^3 x1^3 + 9.0955 x0^2 x1^4 + -0.9588 x0 x1^5 + -9.6904 x1^6 + -0.5283 x0^7 + 0.3958 x0^6 x1 + -1.0637 x0^5 x1^2 + 0.0579 x0^4 x1^3 + 0.2720 x0^3 x1^4 + -0.7079 x0^2 x1^5 + 0.5833 x0 x1^6 + 6.2395 x1^7 + 7.4447 x0^8 + -0.0432 x0^7 x1 + 8.9138 x0^6 x1^2 + 0.4040 x0^5 x1^3 + 7.6487 x0^4 x1^4 + 0.6012 x0^3 x1^5 + 10.9499 x0^2 x1^6 + 1.4716 x0 x1^7 + 20.1573 x1^8 + -9.8061 = 0

Dataset number five:




```
Linear LogisticRegression on dataset 5, train accuracy: 100.00, test accuracy: 100.00
Equation of decision boundary: -11.3585 x0 + -2.2256 x1 + -0.1210 x0^2 + -0.3239 x0 x1 + 0.0520 x1^2 +
```

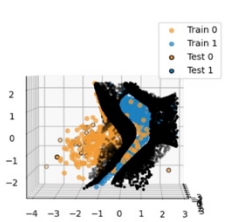
Best chromosome:

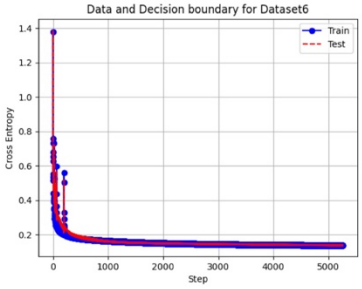
Best chromosome: power=4, n_iter=14705, lr=1.0

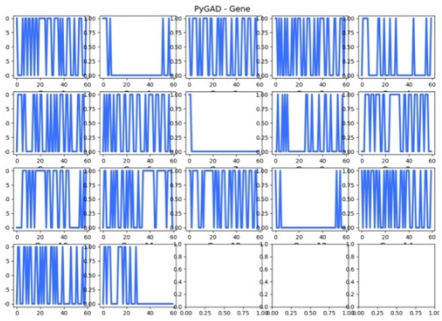
Final equation of decision boundary in full:

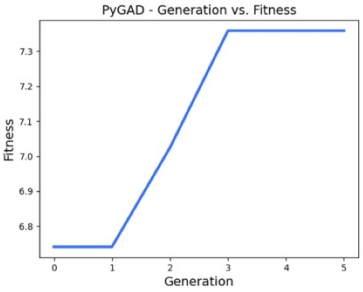
Equation of decision boundary: -11.3585 x0 + -2.2256 x1 + -0.1210 x0^2 + -0.3239 x0 x1 + 0.0520 x1^2 + -0.8577 x0^3 + -0.8096 x0^2 x1 + -0.8899 x0 x1^2 + -2.3785 x1^3 + -0.3391 x0^4 + 0.1692 x0^3 x1 + -0.0280 x0^2 x1^2 + -0.1680 x0 x1^3 + -0.0050 x1^4 + 3.5568 x0^5 + -1.7599 x0^4 x1 + 1.0210 x0^3 x1^2 + -1.0332 x0^2 x1^3 + 0.3193 x0 x1^4 + -2.4168 x1^5 + 0.1495 = 0

Dataset number six:










```
Linear LogisticRegression on dataset 6, train accuracy: 95.28, test accuracy: 95.86
Equation of decision boundary: 0.5394 x0 + 4.5997 x1 + 0.9149 x2 + -0.3867 x0^2 + 0.0384 x0 x1 +
```

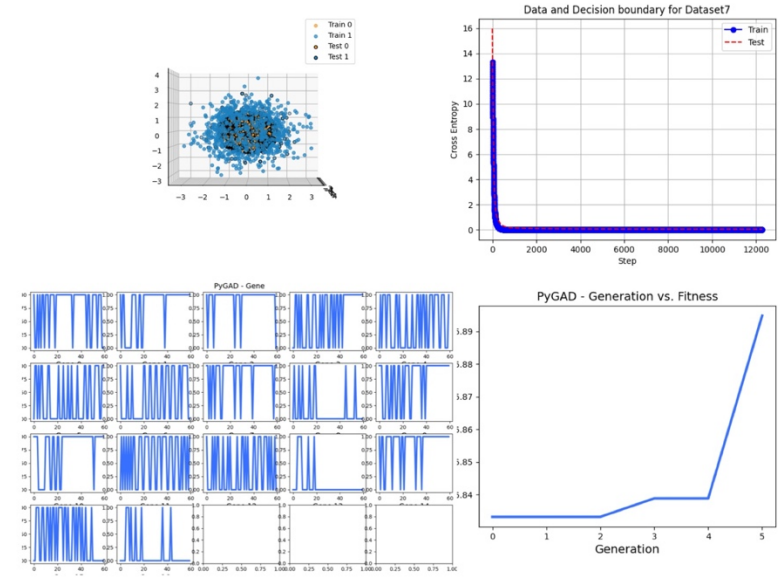
Best chromosome:

Best chromosome: power=4, n_iter=16081, lr=1.0

Final equation of decision boundary in full:

Equation of decision boundary: 0.5394 x0 + 4.5997 x1 + 0.9149 x2 + -0.3867 x0^2 + 0.0384 x0 x1 + 0.8415 x0 x2 + -0.7693 x1^2 + -1.2401 x1 x2 + 4.9968 x2^2 + 0.2485 x0^3 + 1.4268 x0^2 x1 + 0.9842 x0^2 x2 + -0.0485 x0 x1^2 + -0.9992 x0 x1 x2 + -0.8132 x0 x2^2 + 1.6957 x1^3 + -0.2095 x1^2 x2 + 2.3918 x1 x2^2 + -0.0168 x2^3 + 0.0926 x0^4 + 0.0646 x0^3 x1 + 0.0309 x0^3 x2 + -0.1923 x0^2 x1^2 + -0.3822 x0^2 x1 x2 + -0.1419 x0^2 x2^2 + -0.4029 x0 x1^3 + 0.2911 x0 x1^2 x2 + -0.4693 x0 x1 x2^2 + -0.4092 x0 x2^3 + -1.3681 x1^4 + 0.7534 x1^3 x2 + -3.1261 x1^2 x2^2 + 1.3098 x1 x2^3 + -0.8078 x2^4 + -0.0390 x0^5 + 0.0383 x0^4 x1 + -0.0762 x0^4 x2 + -0.0385 x0^3 x1^2 + -0.1341 x0^3 x1 x2 + 0.0742 x0^3 x2^2 + -0.6186 x0^2 x1^3 + 0.3178 x0^2 x1^2 x2 + -0.3131 x0^2 x1 x2^2 + -0.3763 x0^2 x2^3 + 0.0727 x0 x1^4 + 0.5237 x0 x1^3 x2 + -0.7102 x0 x1^2 x2^2 + 0.0245 x0 x1 x2^3 + 0.1390 x0 x2^4 + 0.3680 x1^5 + 0.0523 x1^4 x2 + 0.1263 x1^3 x2^2 + 0.9570 x1^2 x2^3 + -1.4035 x1 x2^4 + 0.0863 x2^5 + -3.0643 = 0

Dataset number seven:




```
Linear LogisticRegression on dataset 7, train accuracy: 100.00, test accuracy: 99.24
Equation of decision boundary: -0.8152 x0 + -0.2290 x1 + -0.0635 x2 + -4.1272 x0^2 + 1.0312 x0 x1 +
```

Best chromosome:

Best chromosome: power=5, n_iter=8305, lr=0.5

Final equation of decision boundary in full:

Equation of decision boundary: -0.8152 x0 + -0.2290 x1 + -0.0635 x2 + -4.1272 x0^2 + 1.0312 x0 x1 + -0.5513 x0 x2 + -3.8010 x1^2 + -0.2726 x1 x2 + -3.6865 x2^2 + -1.1110 x0^3 + -0.0850 x0^2 x1 + -0.1037 x0^2 x2 + 0.0451 x0 x1^2 + -0.0224 x0 x1 x2 + -0.4885 x0 x2^2 + -0.6822 x1^3 + 0.5331 x1^2 x2 + -0.0111 x1 x2^2 + 0.1319 x2^3 + -5.6804 x0^4 + 1.7313 x0^3 x1 + -1.0771 x0^3 x2 + 1.1206 x0^2 x1^2 + 0.3476 x0^2 x1 x2 + 0.9765 x0^2 x2^2 + 0.1914 x0 x1^2 x2 + 0.7194 x0 x1 x2^2 + -0.0129 x0 x2^3 + -5.3521 x1^4 + -0.2956 x1^3 x2 + 1.5791 x1^2 x2^2 + -0.7960 x1 x2^3 + -5.3386 x2^4 + -1.0085 x0^5 + 0.3011 x0^4 x1 + 0.0407 x0^4 x2 + -0.1588 x0^3 x1^2 + -0.2603 x0^3 x1 x2 + -0.7590 x0^3 x2^2 + -0.7634 x0^2 x1^3 + -0.1915 x0^2 x1^2 x2 + 0.3281 x0^2 x1 x2^2 + 0.0538 x0^2 x2^3 + 0.5720 x0 x1^4 + -0.1111 x0 x1^3 x2 + -0.3169 x0 x1^2 x2^2 + 0.1346 x0 x1 x2^3 + -0.3014 x0 x2^4 + -0.8014 x1^5 + 0.6662 x1^4 x2 + -0.2927 x1^3 x2^2 + 0.7493 x1^2 x2^3 + 0.1528 x1 x2^4 + 0.0049 x2^5 + -8.0226 x0^6 + 2.4906 x0^5 x1 + -1.6142 x0^5 x2 + 2.1607

x0^4 x1^2 + 0.6827 x0^4 x1 x2 + 2.2288 x0^4 x2^2 + 0.2795 x0^3 x1^3 + -0.2122 x0^3 x1^2 x2 + 0.7624 x0^3 x1 x2^2 + -0.0276 x0^3 x2^3 + 2.2779 x0^2 x1^4 + 0.2028 x0^2 x1^3 x2 + 1.5455 x0^2 x1^2 x2^2 + -0.3296 x0^2 x1 x2^3 + 1.9556 x0^2 x2^4 + -0.6178 x0 x1^5 + 0.2891 x0 x1^4 x2 + 0.5757 x0 x1^3 x2^2 + 0.4350 x0 x1^2 x2^3 + 0.2895 x0 x1 x2^4 + -0.0781 x0 x2^5 + -7.5402 x1^6 + -0.0201 x1^5 x2 + 2.5893 x1^4 x2^2 + -0.8716 x1^3 x2^3 + 2.7241 x1^2 x2^4 + -0.7745 x1 x2^5 + -8.0035 x2^6 + 1.3001 x0^7 + -0.0480 x0^6 x1 + 1.7745 x0^6 x2 + 0.2916 x0^5 x1^2 + -0.9335 x0^5 x1 x2 + -0.7043 x0^5 x2^2 + -0.6293 x0^4 x1^3 + -0.1202 x0^4 x1^2 x2 + 0.1785 x0^4 x1 x2^2 + 0.4287 x0^4 x2^3 + 0.3329 x0^3 x1^4 + -0.4720 x0^3 x1^3 x2 + -0.2962 x0^3 x1^2 x2^2 + -0.1459 x0^3 x1 x2^3 + -0.7735 x0^3 x2^4 + -0.6191 x0^2 x1^5 + 0.2912 x0^2 x1^4 x2 + -0.0506 x0^2 x1^3 x2^2 + -0.1784 x0^2 x1^2 x2^3 + 0.4359 x0^2 x1 x2^4 + -0.2269 x0^2 x2^5 + 1.0682 x0 x1^6 + -0.2749 x0 x1^5 x2 + -0.1412 x0 x1^4 x2^2 + 0.0979 x0 x1^3 x2^3 + -0.1291 x0 x1^2 x2^4 + 0.4306 x0 x1 x2^5 + 0.6065 x0 x2^6 + -0.1777 x1^7 + 0.4442 x1^6 x2 + 0.0032 x1^5 x2^2 + 0.8982 x1^4 x2^3 + -0.3157 x1^3 x2^4 + 0.3201 x1^2 x2^5 + 0.4906 x1 x2^6 + 0.9466 x2^7 + 4.7867 x0^8 + -1.9557 x0^7 x1 + 0.9261 x0^7 x2 + 7.2818 x0^6 x1^2 + -1.3075 x0^6 x1 x2 + 6.8026 x0^6 x2^2 + -1.2804 x0^5 x1^3 + 0.6273 x0^5 x1^2 x2 + -0.1228 x0^5 x1 x2^2 + 0.6792 x0^5 x2^3 + 5.3191 x0^4 x1^4 + -0.2386 x0^4 x1^3 x2 + 2.3104 x0^4 x1^2 x2^2 + -0.4001 x0^4 x1 x2^3 + 4.8974 x0^4 x2^4 + -0.2146 x0^3 x1^5 + 0.1144 x0^3 x1^4 x2 + 0.2170 x0^3 x1^3 x2^2 + 0.1852 x0^3 x1^2 x2^3 + 0.1110 x0^3 x1 x2^4 + 0.1804 x0^3 x2^5 + 5.8994 x0^2 x1^6 + 0.3299 x0^2 x1^5 x2 + 2.1959 x0^2 x1^4 x2^2 + -0.1361 x0^2 x1^3 x2^3 + 1.9947 x0^2 x1^2 x2^4 + -0.7756 x0^2 x1 x2^5 + 4.6935 x0^2 x2^6 + -0.2478 x0 x1^7 + -0.5184 x0 x1^6 x2 + 0.5490 x0 x1^5 x2^2 + 0.4990 x0 x1^4 x2^3 + 0.3653 x0 x1^3 x2^4 + 0.3222 x0 x1^2 x2^5 + -0.6173 x0 x1 x2^6 + 0.1467 x0 x2^7 + 5.1682 x1^8 + 1.4035 x1^7 x2 + 5.4654 x1^6 x2^2 + -1.0177 x1^5 x2^3 + 4.7867 x1^4 x2^4 + -1.3206 x1^3 x2^5 + 5.3588 x1^2 x2^6 + 1.4599 x1 x2^7 + 5.1412 x2^8 + -9.3498 = 0

As seen in the above results, the logistic regression model trained using the genetic algorithm achieved the best results in each of the datasets.