BIO COMPUTATION

HASSAN RIZVEE

S1900436

1. INTRODUCTION

This study examines how a system handles classification issues with pre-defined anticipated output variables for a particular kind of input variables. The aim was to accurately match the input variables to the output variables for the supplied inputs for randomly generated inputs and outputs in three datasets.

Data set 1 is made up of 5-bit 32 binary strings, whereas data set 2 is made up of 6-bit 64 binary strings. Data set 3 contains 2000 strings of real data consisting of floating values ranging from 0 to 1, with strings to 6 decimal places (each string containing 6 input values and its output). All data sets have input and an output (0 or 1). The goal of the implemented algorithm is to accurately identify the output from the inputs, which defines the method's accuracy.

A rule-based genetic algorithm is particularly successful for the first two data sets. However, with the third data set, this approach proved ineffective, and a neural network must be implemented in the algorithm.

This paper will address experiment research and experimental procedures that are common and distinctive in three data sets. Various graphs versus various parameters will be supplied as talks progress.

2. BACKGROUND RESEARCH

2.1 DATAMINING

As the name implies, data mining is the digging or mining of data to find patterns or more information on how the data can be used efficiently to solve a specific problem. (Sharma, 2022) Data mining is typically used on large sets of data that cannot be classified or evaluated manually. Data mining is essential in this topic of interest because the methodology used to handle the given datasets can be used for data mining.

2.2 Classification

Data is separated and sorted into categories based on similarities through the classification process. In AI, classification picks up patterns from training data to classify new data using what has been learnt (Lucidworks, 2021). Datasets having binary values also have a binary categorization since the data is divided into two categories. Training and testing phases are the two basic classification processes. As was already said, before moving on to the next stage of testing, a pattern must first be discovered using training data. The outcomes of the testing phase would be used to gauge the effectiveness of the categorization algorithm.

2.3 Evolutionary Computing

A class of global optimization algorithms inspired by natural growth is known as "evolutionary computing" (Chanal, Kakkasageri and Manvi, 2021). Natural growth, as previously indicated, entails testing a population of individuals to see whether each member of that population is capable of undergoing the other processes of natural growth or evolution, such as selection, cross-over, and mutation. By forcing each individual to perform the required functions and calculating each generation's fitness, evolutionary computing can evolve a set of data to find its most optimal solution.

3. EXPERIMENTATIONS

A cursory examination of the supplied data sets reveals that the first two data sets can be resolved using the same methodology. The amount of bits in each line and the overall number of lines change between data sets one and two. The third data set has a more complicated appearance because it includes actual float data. Compared to data sets one and two, there are much more rows and columns. The algorithm used to solve the first two data sets cannot be utilized to solve the third data set.

Algorithms were created in the Python language for these experiments because of its libraries and because it was much simpler than other languages. It is simpler to implement graphs, perform calculations, and have the ability to handle large data sets.

3.1 Data Set 1 and Data Set 2

Both of these datasets have commonalities, which may be seen by closely examining them both. Data set 2 is a bigger collection that resembles data set 1 in certain ways. 32 rows of data make up Data Set 1, each with a binary input of 5 bits and a binary output. The second data set consists of 64 rows of data, each of which has a single 6-bit binary input and binary output. Data set 2 distinguishes itself from data set 1 by having more input bits and more data rows. Therefore, the same technique can be used to handle these two data sets by changing parameters like the number of genes and the size of a gene.

The holdout method was used to test the classifiers in the two data sets. The ratios of the training set and the testing set were then checked to see if there was a difference.
In the algorithm, The first data is collected from a CSV file and then it is split into two columns. To test the data set, set the training ratio and split the data into two sets. We then generate the initial population using rules. The population gets progressively bigger. It first calculates the fitness of the individuals in the population by comparing it with the training data set. Then it starts the selection process. The method used for the selection process is the tournament selection method and the Roulette Wheel Selection. Parents will be re-selected during the crossover procedure. Offspring produced during the crossover phase are incorporated into the population. This algorithm will have the capability of crossover in single or double points. Depending on the selected crossover approach, parameters must be changed. Following crossing, each person's fitness will be assessed once again. And now, whenever it is feasible, mutation will be used. The method of mutation is bit flipping. The algorithm won't be able to attain and stop at a local optimum if you mutate. There will be more modified individuals in the population. The final step is to replace the current population with the new one that was created during this process.

This method is continued until the program's maximum generation is reached. When the maximum number of generations has been achieved, a graph is generated to represent the score across the number of generations. To aid comprehension, the application will draw three lines in each graph to depict the worst, highest, and average performance.

There are parameters that can be fine-tuned to improve the performance of this method. Maximum generation can be changed to modify the number of generations that this algorithm progresses through. If the population size has to be modified, the number of persons in the population can be adjusted. The holdout method ration may be changed to vary the train-to-test ratio. The crossover technique can be adjusted to evaluate whether there is a performance difference between a single point crossover and a double point crossover. There is also a mutation rate that may be changed to fine-tune the algorithm's performance.

Holdout Method Ratio to 0.80, single point crossover, number of generations to 500, initial population size to 20, and mutation rate to 0.2 are the default settings evaluated with datasets 1 and 2. Results for the initial run against datasets 1 and 2 are shown in Figures 1 and 2, respectively.

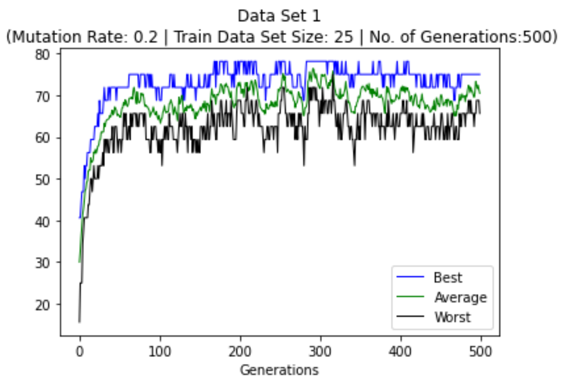


Figure 1

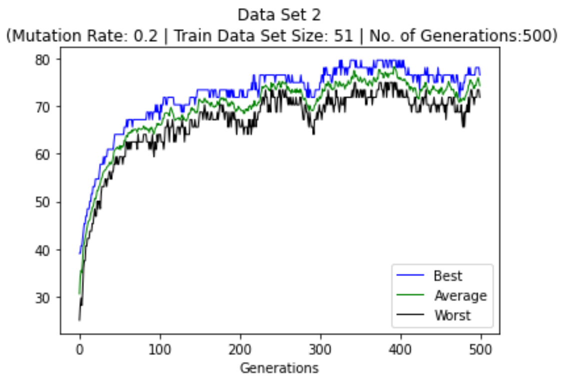


Figure 2

After population and generational totals were adjusted, this program's settings are currently being evaluated to see whether there is any difference in how well it works.

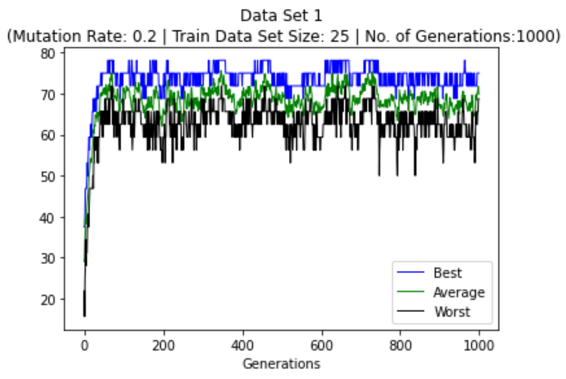


Figure 3

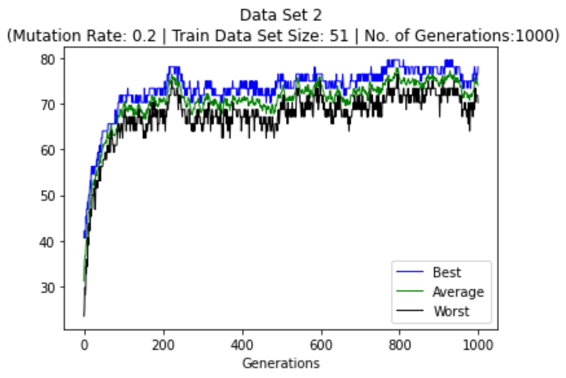


Figure 4

The graphs of datasets 1 and 2 after increasing the population to 20 and the total number of generations to 1000 are shown in Figures 3 and 4. Although the algorithm's general consistency has improved as a result of this update, its fitness has not significantly increased.

The second experiment involved comparing the single point crossover and double point crossover parts of the algorithm to determine whether there were any differences. Only the crossover type was changed in this regard, not any other factors.

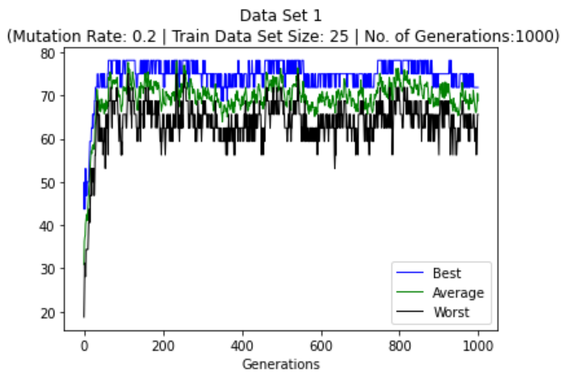


Figure 5

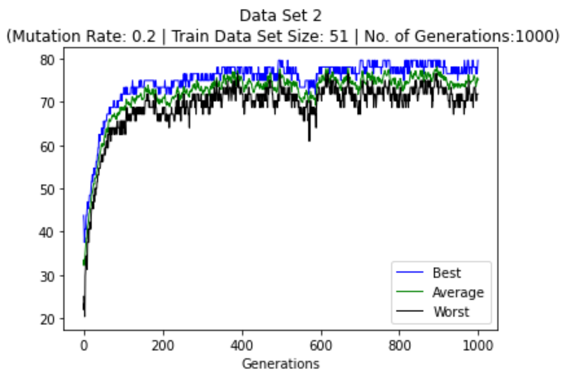


Figure 6

There were no obvious results from this experiment. The algorithm's overall performance on single-point or double-point crossovers remained unchanged.

The purpose of the last experiment is to evaluate the effects of modifying the algorithm's mutation parameter. All of the aforementioned studies were conducted using a mutation rate of 0.2. After raising the mutation rate to 0.4 and subsequently 0.7, we will now observe differences in performance.

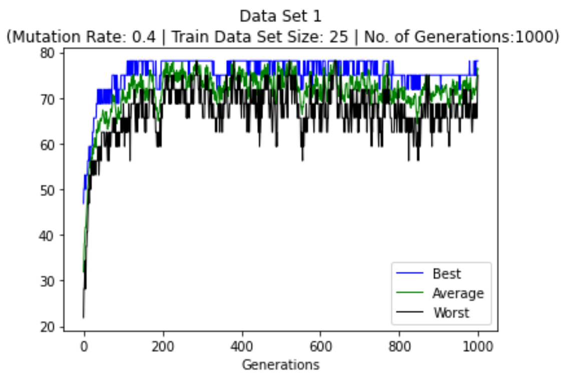


Figure7

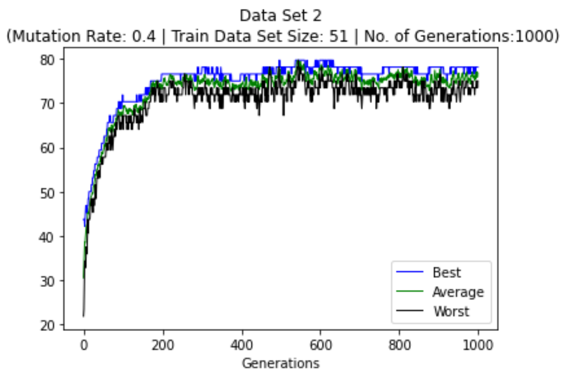


Figure8

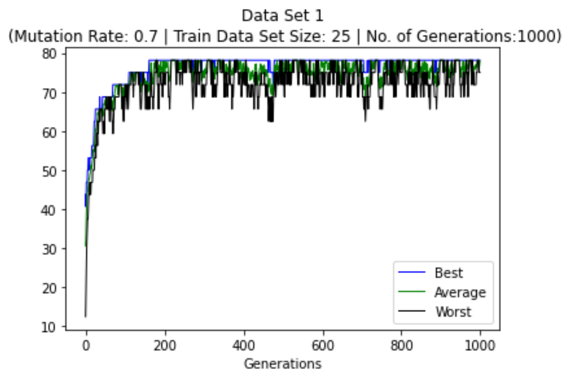


Figure 9

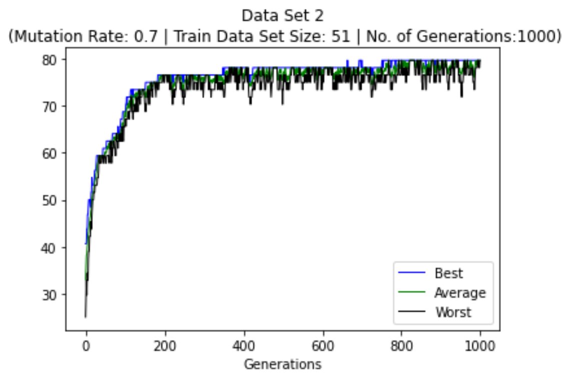


Figure 10

The algorithm's performance for datasets 1 and 2 after adjusting the mutation rate to 0.4 is shown in Figures 7 and 8 above. Figures 9 and 10 above illustrate the algorithm's performance after further increasing mutation to 0.7. After adjusting all the parameters that may be used to fine-tune the algorithm, it can be inferred from this that the algorithm cannot go over 80%. This is most likely caused by the algorithm being forced to deal with local optima, which prevents it from improving further.

Before coming to a final conclusion on this, another experiment was run with a larger population and more generations. This is done to determine whether performance improves if additional generations are allowed to run. However, as seen in Figures 11 and 12, there was no improvement that made it evident that the algorithm had found its local optimum.

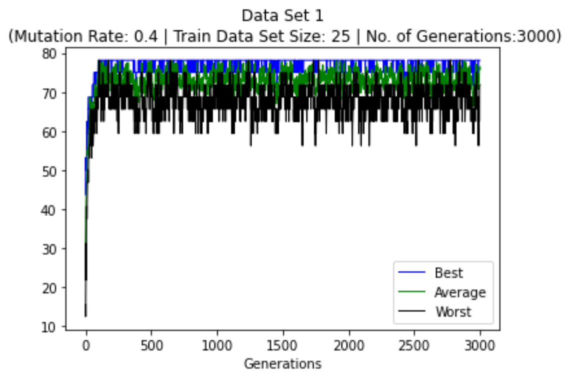


Figure 11

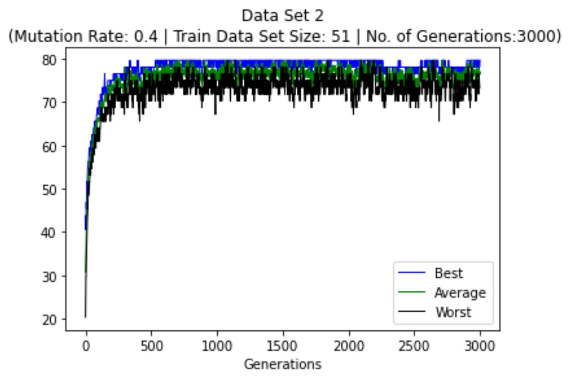


Figure 12

3.2 Data Set 3

Data set 3 contains actual float value data, as contrast to the previous tested data set. Thus, the previously employed method cannot be used to solve this dataset. Each row of data (a total of 2000 rows) has 1 binary output and 6 floating values as input.

The prototype may be trained using a two-layered neural network (NN). The primary reason a neural network was used for this data set is that neural network layers are able to investigate various types of data sets since they are not restricted to any one type of data set.

A NN is trained by making several unsuccessful efforts to answer the problem. A random weight and bias vector is assigned at the beginning, and attempts are performed. Predictions from NN are compared to the predicted results at the end of each step or iteration. After then, changes are made to improve NN's accuracy. This procedure is repeated until the results are most similar to what was anticipated. In order to prevent occurrences of overfitting or underfitting, it is crucial to define the parameters for learning rate and iterations.

With dataset 3, the default parameters are a training set to test set ratio of 0.80, 20,000 iterations, and a learning rate of 0.02. Results for the initial run against dataset 3 with default parameters are shown in Figure 13.

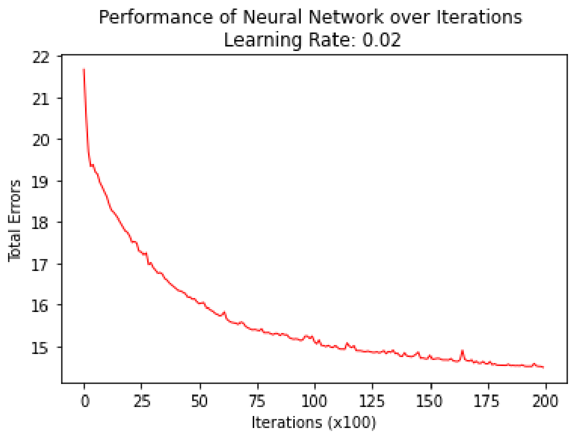


Figure 13

Based on the first run, it appears that there is room for improvement, since the proportion of mistakes is 14.50%. Another experiment was conducted by increasing the number of repetitions from 20,000 to 200,000. Figure 14 depicts the outcome of this adjustment.

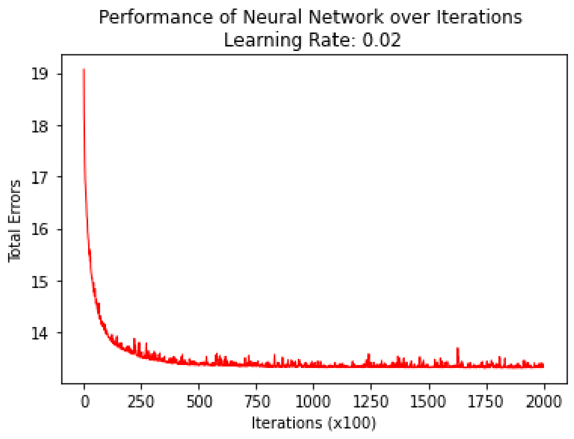


Figure 14

The results show that NN has improved throughout the iterations. By increasing the number of iterations from 20,000 to 200,000, the percentage of errors was lowered from 14.50 to 13.55.

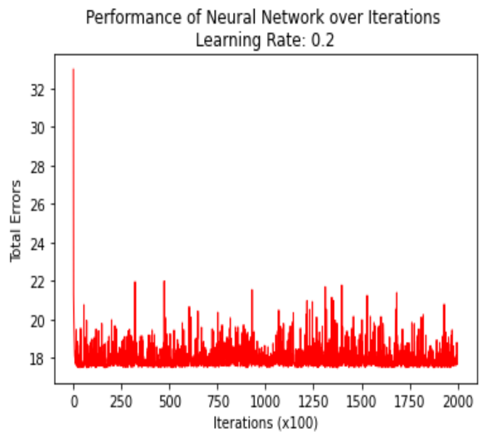


Figure 15

Another test was carried out by increasing the mutation rate from 0.02 to 0.2, and the results are shown in Figure 15 above.

It was discovered that raising the learning rate resulted in less consistency and lower performance. This is seen in Figure 15.

4. Conclusion

I worked on three data sets for this course. The first two data sets were addressed using a genetic algorithm , while the third was solved using a neural network.
Through these three experiments, I discovered that Genetic algorithm works best on linear and simple data sets, but Neural network performs better on complicated and huge data sets. Neural network is easy to implement.

5. Reference

Sharma, R. (2022) Classification in Data Mining explained: Types, Classifiers & Applications [2022] Available from : Classification in Data Mining Explained: Types, Classifiers & Applications [2022] | upGrad blog. [Accessed 20 May 2022].

Lucidworks (2021) Clustering vs. Classification in AI: How Are They Different? Available from: Video: Clustering vs. Classification in AI | Lucidworks. [Accessed 17 May 2022]

Chanal, P. M., Kakkasageri, M. S. and Manvi, S. K. S. (2021) Recent Trends in Computational Intelligence Enabled Research. Cambridge, Massachusetts: Academic Press. Science Direct[online]. Available from: Security and privacy in the internet of things: computational intelligent techniques-based approaches - ScienceDirect [Accessed 20 May 2022].

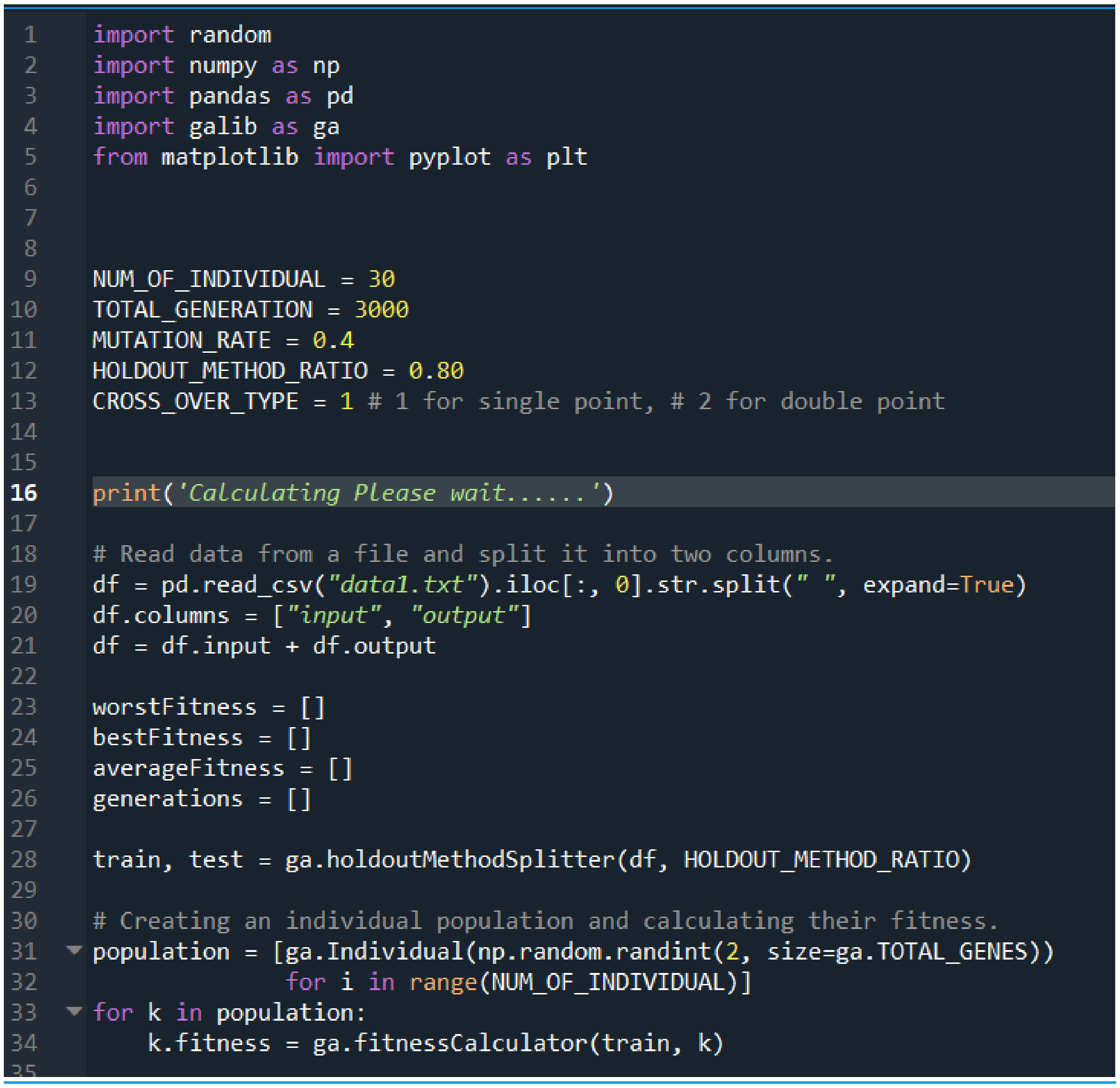
Bajpai, P. and Kumar, M. (2010) Genetic Algorithm – an Approach to Solve Global Optimization Problems. Indian Journal of Computer Science and Engineering [online]. 1 (3), pp. 199-206. [Accessed 17 May 2022].

Déborah Mesquita (No date) Python AI: How to Build a Neural Network & Make Predictions. Available from: Python AI: How to Build a Neural Network & Make Predictions – Real Python [Accessed 24 May 2022].

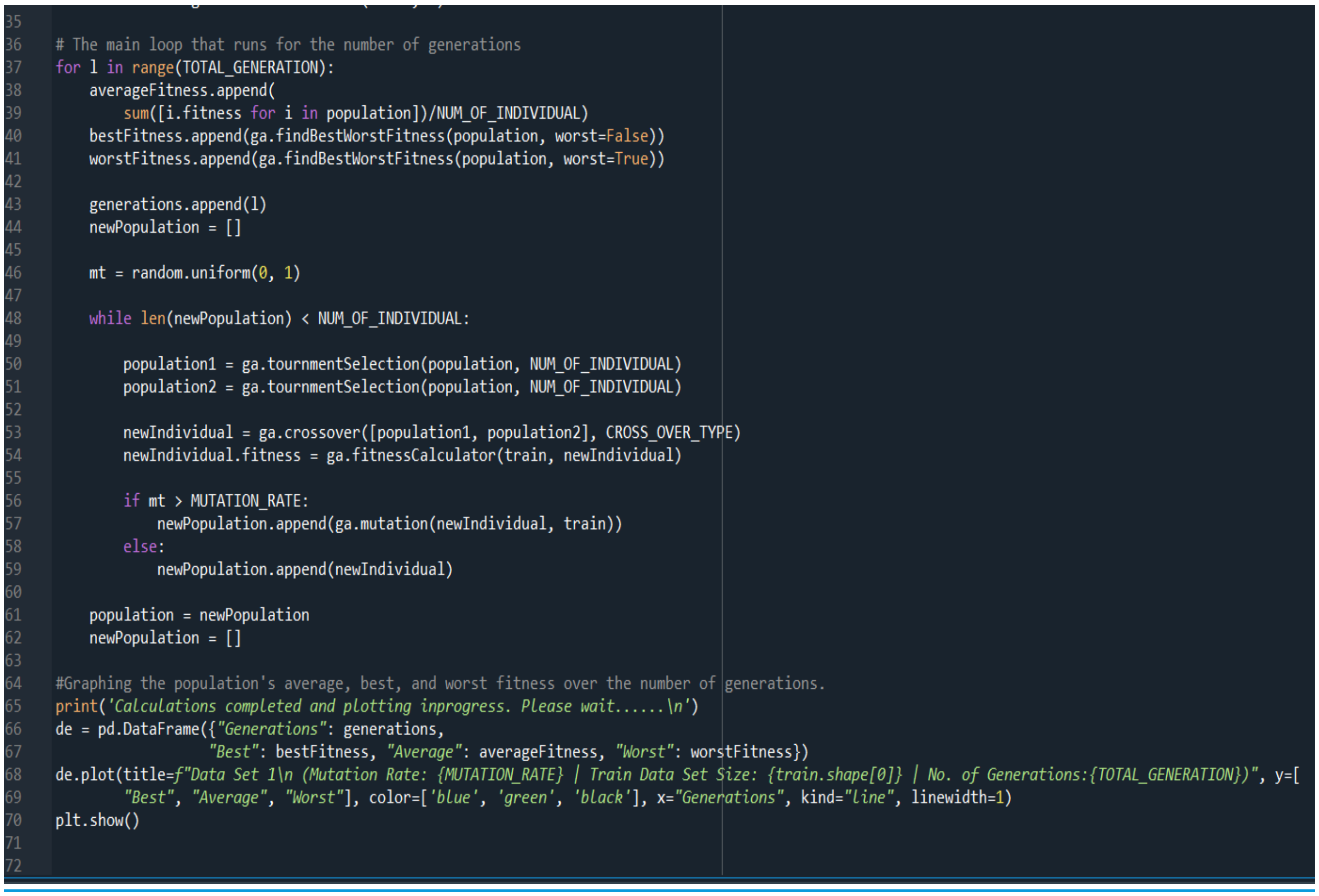
Karcioğlu, A.A. and Bulut, H. (2021) Performance Evaluation of Classification Algorithms Using Hyperparameter Optimization [online]. pp. 354- 358. [Accessed 19 May 2022].

6. Source code

Data set 1

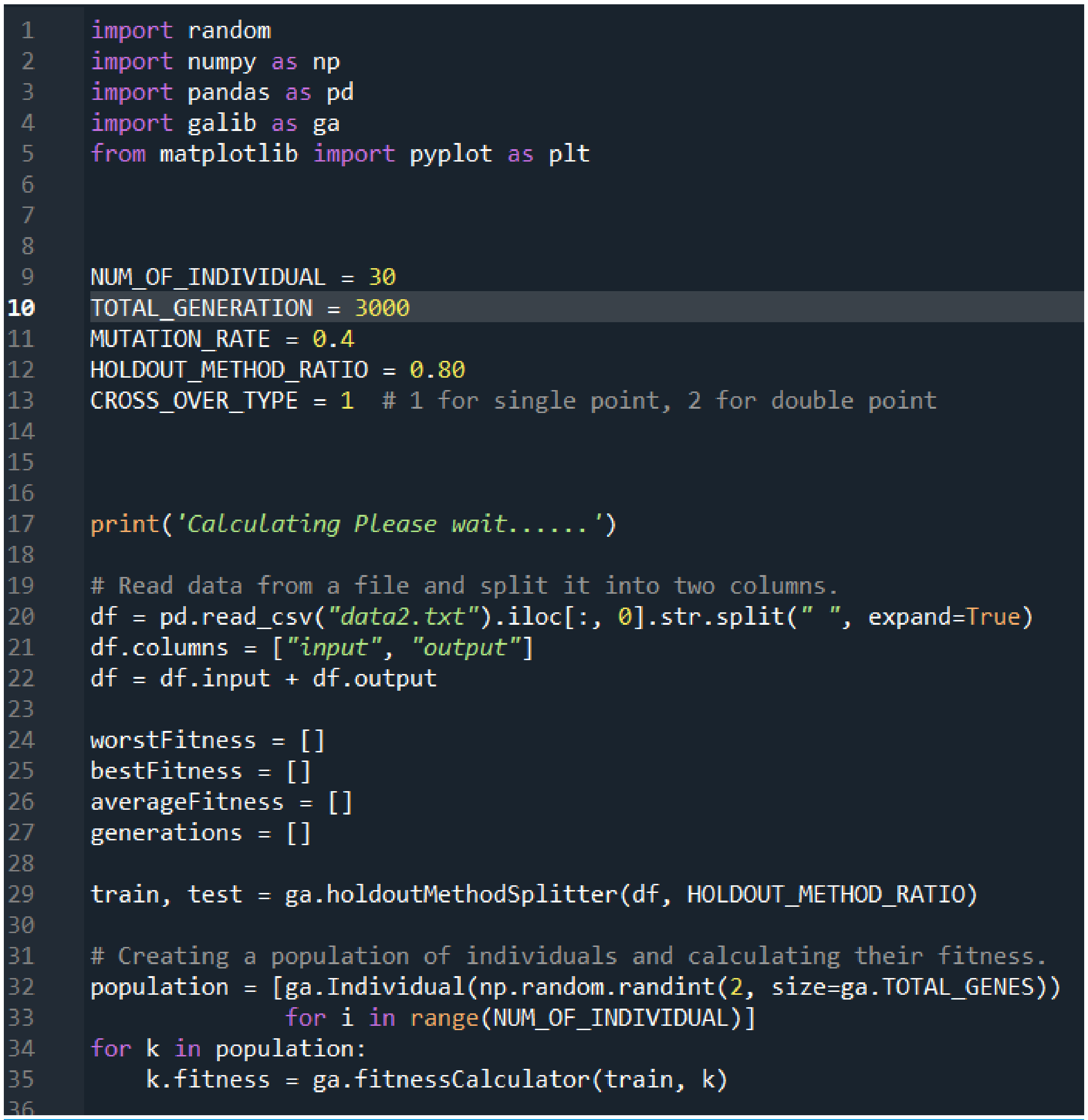


```python
import random
import numpy as np
import pandas as pd
import galib as ga
from matplotlib import pyplot as plt



NUM_OF_INDIVIDUAL = 30
TOTAL_GENERATION = 3000
MUTATION_RATE = 0.4
HOLDOUT_METHOD_RATIO = 0.80
CROSS_OVER_TYPE = 1 # 1 for single point, # 2 for double point


print('Calculating Please wait......')

# Read data from a file and split it into two columns.
df = pd.read_csv("data1.txt").iloc[:, 0].str.split(" ", expand=True)
df.columns = ["input", "output"]
df = df.input + df.output

worstFitness = []
bestFitness = []
averageFitness = []
generations = []

train, test = ga.holdoutMethodSplitter(df, HOLDOUT_METHOD_RATIO)

# Creating an individual population and calculating their fitness.
population = [ga.Individual(np.random.randint(2, size=ga.TOTAL_GENES))
              for i in range(NUM_OF_INDIVIDUAL)]
for k in population:
    k.fitness = ga.fitnessCalculator(train, k)
```

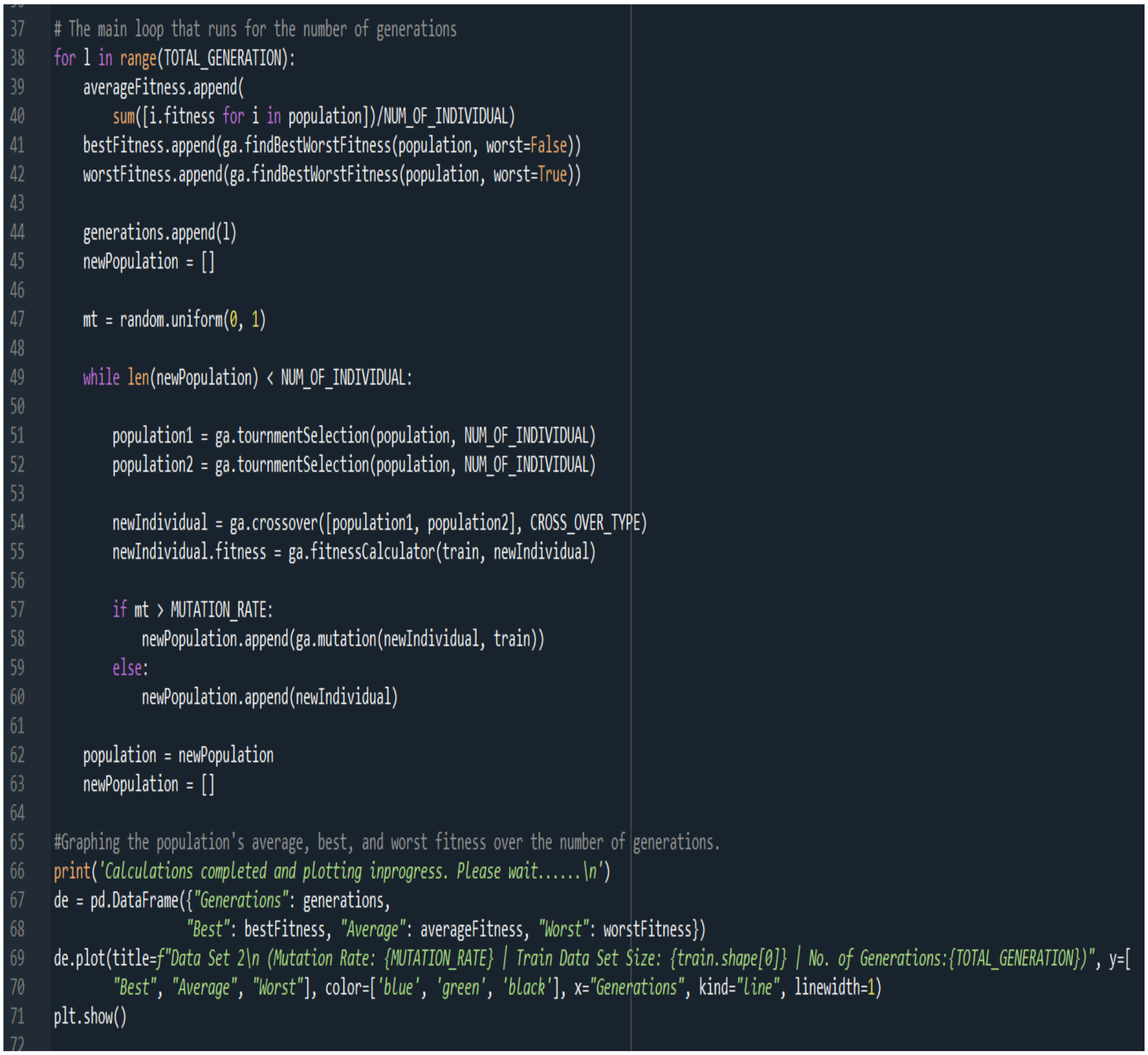


```python

# The main loop that runs for the number of generations
for l in range(TOTAL_GENERATION):
    averageFitness.append(
        sum([i.fitness for i in population])/NUM_OF_INDIVIDUAL)
    bestFitness.append(ga.findBestWorstFitness(population, worst=False))
    worstFitness.append(ga.findBestWorstFitness(population, worst=True))

    generations.append(l)
    newPopulation = []

    mt = random.uniform(0, 1)

    while len(newPopulation) < NUM_OF_INDIVIDUAL:

        population1 = ga.tournmentSelection(population, NUM_OF_INDIVIDUAL)
        population2 = ga.tournmentSelection(population, NUM_OF_INDIVIDUAL)

        newIndividual = ga.crossover([population1, population2], CROSS_OVER_TYPE)
        newIndividual.fitness = ga.fitnessCalculator(train, newIndividual)

        if mt > MUTATION_RATE:
            newPopulation.append(ga.mutation(newIndividual, train))
        else:
            newPopulation.append(newIndividual)

    population = newPopulation
    newPopulation = []

#Graphing the population's average, best, and worst fitness over the number of generations.
print('Calculations completed and plotting inprogress. Please wait......\n')
de = pd.DataFrame({"Generations": generations,
                  "Best": bestFitness, "Average": averageFitness, "Worst": worstFitness})
de.plot(title=f"Data Set 1\n (Mutation Rate: {MUTATION_RATE} | Train Data Set Size: {train.shape[0]} | No. of Generations:{TOTAL_GENERATION})", y=[
        "Best", "Average", "Worst"], color=['blue', 'green', 'black'], x="Generations", kind="line", linewidth=1)
plt.show()


```

Data set 2

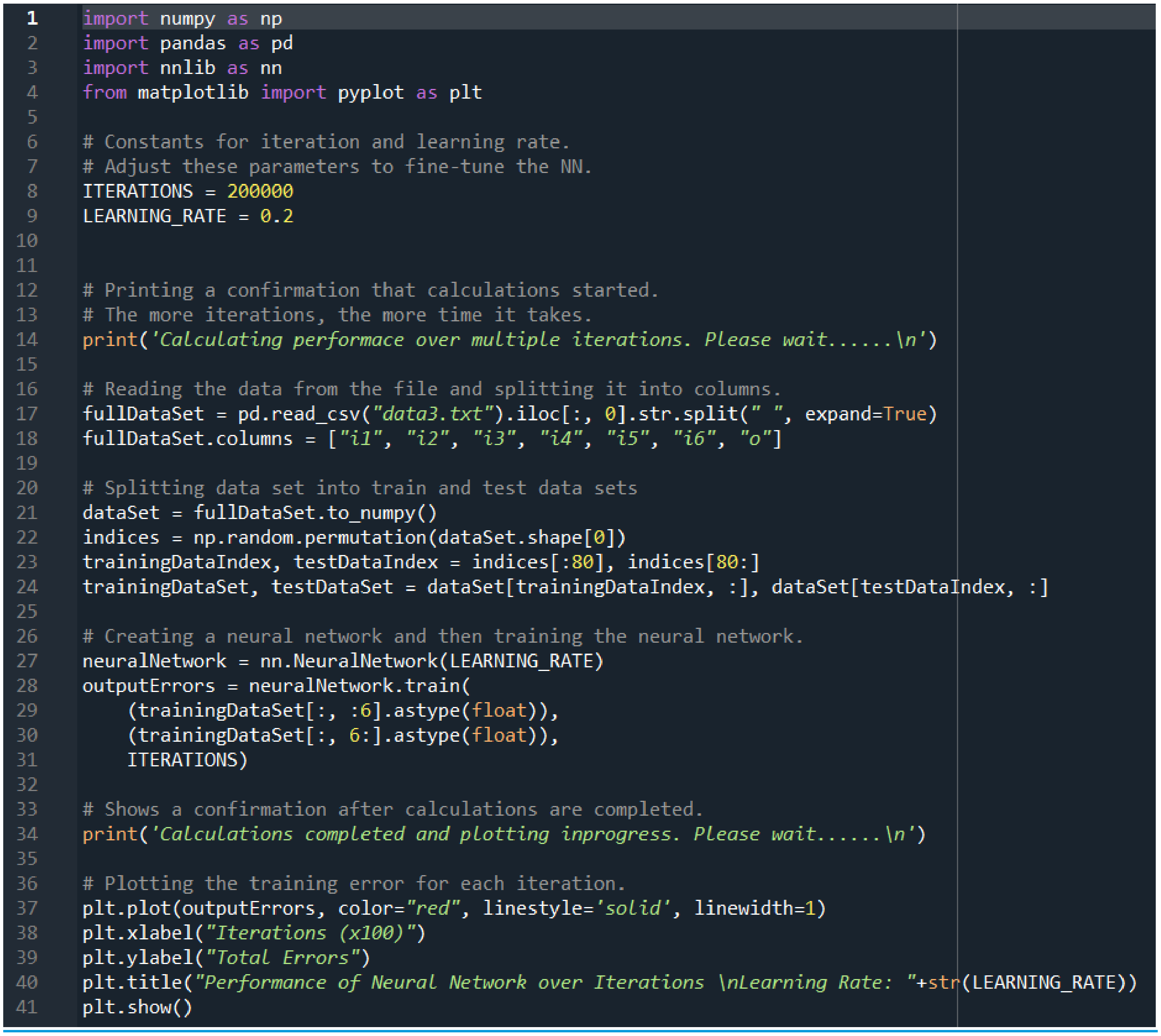


```python
import random
import numpy as np
import pandas as pd
import galib as ga
from matplotlib import pyplot as plt



NUM_OF_INDIVIDUAL = 30
TOTAL_GENERATION = 3000
MUTATION_RATE = 0.4
HOLDOUT_METHOD_RATIO = 0.80
CROSS_OVER_TYPE = 1  # 1 for single point, 2 for double point



print('Calculating Please wait......')

# Read data from a file and split it into two columns.
df = pd.read_csv("data2.txt").iloc[:, 0].str.split(" ", expand=True)
df.columns = ["input", "output"]
df = df.input + df.output

worstFitness = []
bestFitness = []
averageFitness = []
generations = []

train, test = ga.holdoutMethodSplitter(df, HOLDOUT_METHOD_RATIO)

# Creating a population of individuals and calculating their fitness.
population = [ga.Individual(np.random.randint(2, size=ga.TOTAL_GENES))
              for i in range(NUM_OF_INDIVIDUAL)]
for k in population:
    k.fitness = ga.fitnessCalculator(train, k)
```



```python
# The main loop that runs for the number of generations
for l in range(TOTAL_GENERATION):
    averageFitness.append(
        sum([i.fitness for i in population])/NUM_OF_INDIVIDUAL)
    bestFitness.append(ga.findBestWorstFitness(population, worst=False))
    worstFitness.append(ga.findBestWorstFitness(population, worst=True))

    generations.append(l)
    newPopulation = []

    mt = random.uniform(0, 1)

    while len(newPopulation) < NUM_OF_INDIVIDUAL:

        population1 = ga.tournmentSelection(population, NUM_OF_INDIVIDUAL)
        population2 = ga.tournmentSelection(population, NUM_OF_INDIVIDUAL)

        newIndividual = ga.crossover([population1, population2], CROSS_OVER_TYPE)
        newIndividual.fitness = ga.fitnessCalculator(train, newIndividual)

        if mt > MUTATION_RATE:
            newPopulation.append(ga.mutation(newIndividual, train))
        else:
            newPopulation.append(newIndividual)

    population = newPopulation
    newPopulation = []

#Graphing the population's average, best, and worst fitness over the number of generations.
print('Calculations completed and plotting inprogress. Please wait......\n')
de = pd.DataFrame({"Generations": generations,
                   "Best": bestFitness, "Average": averageFitness, "Worst": worstFitness})
de.plot(title=f"Data Set 2\n (Mutation Rate: {MUTATION_RATE} | Train Data Set Size: {train.shape[0]} | No. of Generations:{TOTAL_GENERATION})", y=[
        "Best", "Average", "Worst"], color=['blue', 'green', 'black'], x="Generations", kind="line", linewidth=1)
plt.show()
```

Data set 3



```python
import numpy as np
import pandas as pd
import nnlib as nn
from matplotlib import pyplot as plt

# Constants for iteration and learning rate.
# Adjust these parameters to fine-tune the NN.
ITERATIONS = 200000
LEARNING_RATE = 0.2


# Printing a confirmation that calculations started.
# The more iterations, the more time it takes.
print('Calculating performace over multiple iterations. Please wait......\n')

# Reading the data from the file and splitting it into columns.
fullDataSet = pd.read_csv("data3.txt").iloc[:, 0].str.split(" ", expand=True)
fullDataSet.columns = ["i1", "i2", "i3", "i4", "i5", "i6", "o"]

# Splitting data set into train and test data sets
dataSet = fullDataSet.to_numpy()
indices = np.random.permutation(dataSet.shape[0])
trainingDataIndex, testDataIndex = indices[:80], indices[80:]
trainingDataSet, testDataSet = dataSet[trainingDataIndex, :], dataSet[testDataIndex, :]

# Creating a neural network and then training the neural network.
neuralNetwork = nn.NeuralNetwork(LEARNING_RATE)
outputErrors = neuralNetwork.train(
    (trainingDataSet[:, :6].astype(float)),
    (trainingDataSet[:, 6:].astype(float)),
    ITERATIONS)

# Shows a confirmation after calculations are completed.
print('Calculations completed and plotting inprogress. Please wait......\n')

# Plotting the training error for each iteration.
plt.plot(outputErrors, color="red", linestyle='solid', linewidth=1)
plt.xlabel("Iterations (x100)")
plt.ylabel("Total Errors")
plt.title("Performance of Neural Network over Iterations \nLearning Rate: "+str(LEARNING_RATE))
plt.show()
```