Report on Income Level Classification Using Genetic Programming on the Adult Dataset

Daniel Maguire
Student Number: 23222425
Presentation Link: CS6271 Evolutionary Computation Project Presentation

Introduction

The goal of this project was to classify individuals based on their income levels using the Adult dataset, a collection of demographic and economic data. I am using Genetic Programming (GP) to predict whether an individual's income exceeds $50,000 annually. This task is not just a classification problem but also an investigation into the effectiveness of GP at classification tasks.

First, we'll take a look at the data cleanup and pre-processing work done before moving on to reviewing the fitness configuration, as well as parameters and functionality supplied to the model. Then we'll have a look at core features of the GP algorithm and model functionality. Finally, we'll have a look at the results before exploring my conclusions.

Data Preprocessing

Dataset

The Adult dataset includes the features **age, workclass, education, marital status, relationship, race, sex, capital gain, capital loss, hours worked per week,** and **native country**. Each of these features has the potential to influence the income level.

A separate unseen test data subset was created from the Adult dataset to assess the model's accuracy, to make sure that the model's performance was not just tailored to the training data. I experimented with different splits between the training and test data but I found that an 80:20 train/test split or a 70:30 split allowed for a large enough training set while still providing enough validation data to accurately evaluate the model performance. I ran my final tests with both 80:20 and 70:30 splits.

Handling Missing Values

In addressing missing values in the dataset, I chose to remove rows with these missing values. I made this decision based on the idea that keeping the data complete and accurate is most important. I am assuming that removing these rows won't significantly affect the dataset since the amount of data lost is small.

Feature Engineering

1. **Net Capital**:

A 'Net Capital' feature was developed by subtracting capital losses from capital gains. This was aimed at providing a clearer understanding of an individual's financial status. Tests indicated that the model performed better when using this consolidated 'Net Capital' feature, rather than maintaining capital gains and losses as separate variables.

2. **Age Grouping** (-1, 1, 2):

The age data was categorised into three groups: young (-1), adult (1), and retired (2). This categorization was designed to capture the varying influences of age on income across different life stages. Grouping ages simplified the data, making it easier for the model to interpret these differences.

Normalisation

The StandardScaler was used for numerical columns '**hours-per-week**' and **'net_capital'**. The goal was to ensure that these features are standardised, meaning they would have a mean of zero and a standard deviation of one. This approach helps to prevent any potential bias that might arise due to the different magnitudes of the original numerical values, ensuring a more balanced and accurate model performance. Such a technique is a critical step in maintaining the integrity and effectiveness of the analysis.

Categorical Encoding

Originally, I was using one-hot encoding for the categorical features but I wanted to see if other encoding methods may result in improved accuracy on the test set, so after researching a few other methods I decided to use **Target** encoding and **CatBoost** encoding. I have created one setup with **Target** and the other with **CatBoost** encoding.

The encoding method to use is specified through the **`ENCODING_METHOD`** parameter.

Target Encoding can capture more information per feature, but also it risks overfitting. So I was a bit unsure about this one, but it did perform well and in general it seemed to perform better than the one-hot encoder.

CatBoost Encoding is particularly good at processing categorical data and generally leads to better performance, so this one was very interesting to try and use.

Final Steps in Preprocessing

Exclusion of Original Age Column: After creation of **`age_group`**, the original **`age`** column is excluded to avoid redundancy. I tested both and I was consistently getting better results after removing the original age column.

Preprocessing Output

The function returns the processed DataFrame, along with the scaler and encoder, to ensure consistency in processing future data.

GP Development and Training

Fitness Function

In the development of the fitness function for the GP model, the following methodologies and rationales were applied:

Fitness Function Implementation:

The function processes each row of the encoded training data **`X_train_encoded`**, applying the GP individual to generate predictions. This method ensures a comprehensive evaluation of the model's ability to predict.

Handling Invalid Outputs:

Infinite and Not-a-Number Values: The fitness function checks for and counts **`inf`** and **`nan`** values in the predictions. These values often result from invalid mathematical operations like division by zero and disrupt the learning process.

Threshold Application: If invalid outputs exceed 5% of the training data, a large penalty **`TYPE_ERROR_PENALTY`** is applied. This rule helps penalise poor solutions but allows some room for occasional errors.

Binary Classification Decision:

Predicted values are converted to binary using a 0.5 threshold. I am doing this because the model aims to predict a binary outcome. Converting continuous outputs to binary makes them easier to compare with the target variable.

Accuracy Calculation:

Accuracy is measured by the proportion of correct predictions from all valid predictions, excluding those with **`inf`** or **`nan`** values. This method aligns well with the model's goal to maximise correct income predictions and provides a straightforward way to assess GP individuals' performance.

Type Error Handling:

The function also counts and filters errors similar to the handling of **`inf`** and **`nan`** values. This is to make sure the model can handle outputs that don't match the expected data type, increasing the reliability of the fitness evaluation.

Function Set

While developing this model, I tried a few different function sets, starting originally with only arithmetic, trigonometric, and exponential operations. Using these alone my results were decent, but I wanted to expand the models computational capabilities, so I added a conditional operator to allow decision making and to go along with that I also added comparison and logical operators. Finally, I wanted to see how the model would perform if I gave it looping capabilities, so to achieve this I added a for loop function.

Arithmetic Operations (add, sub, mul,protectedDiv)

These basic arithmetic operations enable the GP individuals to perform fundamental numerical manipulations, which are essential for any mathematical modelling.

```
def protectedDiv(left, right):
    try:
        return left / right
    except ZeroDivisionError:
        return -100000
```

Trigonometric Functions (psin, pcos)

These functions provide the capability to model periodic patterns which might be present in the data.

```
def psin(n):
    try:
        return numpy.sin(n)
    except Exception:
        return numpy.nan

def pcos(n):
    try:
        return numpy.cos(n)
    except Exception:
        return numpy.nan
```

Exponential and Power Functions (pexp, pow2)

These allow the model to incorporate exponential growth or decay patterns. They can help model complex behaviours including nonlinear relationships, which are often found in real-world data.

```
def pow2(n):
    return operator.pow(n, 2)

def pexp(n):
    try:
        return numpy.exp(n)
    except Exception:
        return numpy.nan
```

Conditional Operator (if_then_else)

Adding if else functionality allows the individuals to have a decision-making tool. This function adds a fundamental control flow mechanism which allows GP individuals to make conditional decisions based on the data.

```
def if_then_else(condition, out1, out2):
    return out1 if condition else out2
```

Comparison Operators (gt, lt)

Allows the GP individuals to compare values, adding an essential aspect of decision making and essential to have with a conditional operator.

```
def gt(a, b):
    return a > b


def lt(a, b):
    return a < b
```

Logical Operators (logical_and, logical_or, logical_not)

This function set enables the construction of logical expressions and conditions. These are essential in the decision making processes in computational models.

```
def logical_and(a, b):
    return a and b


def logical_or(a, b):
    return a or b


def logical_not(a):
    return not a
```

Loop (for_loop)

I added a for loop to add iterative capability to the GP individuals. This simulates loop-like behaviour, potentially allowing the model to perform repetitive tasks or aggregate operations, which could potentially help in revealing complex patterns within the dataset.

By using a range of arithmetic, logical and control operations along with more advanced logic like **if_then_else** and **for_loop**, I have provided the GP model a large set of tools to evolve robust and versatile solutions. This diverse selection allows the GP to explore a wide range of potential solutions which increases the likelihood of discovering effective models for predicting income levels.

```
def for_loop(operation, data, iterations):
    result = data
    try :
        iterations = int(iterations)
    except Exception:
        iterations = 1

    for i in range(iterations):
        result = operation(result)
    return result
```

GP Model Parameters

Population Size: 100

A population size of 100 has a balance between diversity of solutions and computational manageability. Larger populations do offer more diversity but at the cost of increased computational resources.

Maximum Generations: 100

Setting max generations to 100 provides enough iterations for the population to evolve and converge while preventing excessive computation.

Crossover Probability (p_crossover_values): [0.4, 0.6, 0.7]

Different crossover probabilities were chosen to see how they affect the model. Lower values like 0.4 and 0.6 lead to fewer changes in each generation, which helps keep the evolution process stable. Higher values like 0.8 and 0.9 mix the genetic information more, which could help find the best solutions faster. This range from moderate to high crossover rates lets us see how changing the frequency of crossover impacts the model's ability to find good solutions. I originally wanted to test crossover values of 0.8 and 0.9, but due to time constraints I tested a reduced set.

Mutation Probability (p_mutation_values): [0.01, 0.1, 0.3]

The chosen mutation probabilities are meant to find a balance between bringing new traits into the population and avoiding too much randomness. A low mutation rate, like 0.01, changes the population slowly and keeps things stable. A medium rate, like 0.1, introduces new traits at a reasonable pace. The highest rate, 0.3, tests how the model behaves with more frequent changes. This range helps us understand how different levels of mutation affect the model’s ability to explore new solutions and improve over time.

Hall of Fame Size: 10

Maintaining a Hall of Fame of the top 10 individuals helps in preserving the best solutions across generations and runs. I found that 10 was best in this case.

Fitness and Evaluation Parameters

MAX_FITNESS: Set at 1.0 to define the upper bound of fitness.

Penalties (NAN_PENALTY, INF_PENALTY, TYPE_ERROR_PENALTY): Set to -1 to penalise invalid outputs effectively.

Tree Complexity Parameters

MIN_TREE_HEIGHT, MAX_TREE_HEIGHT, **LIMIT_TREE_HEIGHT, MUT_MIN_TREE_HEIGHT, MUT_MAX_TREE_HEIGHT**: I kept everything here standard. Any experimenting I did with these values resulted in negative performance so due to time constraints I did not investigate further. I may do so in the future.

Validation and Termination Criteria

VALIDATION_ERROR_DELTA_THRESHOLD: Different thresholds for the first run and subsequent runs are used to adjust the model's sensitivity to validation errors. I am using two different thresholds, one for the first run and one for all other runs because as the runs continue the delta increases, and I found keeping the first run strict while allowing a slightly more lenient threshold for all other runs worked well.

MIN_VALIDATION_ERROR_THRESHOLD: Set at 0.75 to ensure a minimum acceptable level of model accuracy. I was interested in seeing results with an accuracy of at least 0.75.

N_RUNS and N_GUESSES: These parameters control the number of evolutionary runs and guesses, allowing multiple attempts to find the optimal solution. In this implementation N_RUNS controls the number of times we pass the HOF to a new population to continue to evolve them. N_GUESSES controls how many times we test a specific setup. This is required because we want to try and give each setup a chance to perform, and due to the stochastic nature of these algorithms we need to run multiple times to get a better representation of the search space.

Overview of the GP Algorithm

Hall of Fame

Using to ensure that the best solutions aren't lost over generations and runs. We are also using the HOF to keep track of the best individuals over multiple runs.

Multiple Runs and Convergence Check

The algorithm is run multiple times and checks for significant improvements. This helps in avoiding unnecessary computations by stopping the runs if there is no substantial improvement seen.

Validation of the Best Solution

After evolution, the best solution is tested on unseen data. We need to do this to make sure the model works well on unseen data as well as the training data.

Adaptive Termination Based on Validation Error

The algorithm stops if the solution starts to get worse or is not improving enough. This is to try to prevent the model fitting the training data too closely and to save time by getting to optimal solutions faster.

Visualisation and Result Tracking

The fitness evolution is plotted for a visual understanding of the process.

Rationale Behind Decisions

Different Thresholds for Validation Error

Using varied thresholds for different runs accounts for the evolving nature of the model. It's a way to balance between finding a good model and not overfitting.

Exiting Runs on Decreasing Validation Error

If the model starts performing worse on new data, it's a sign to stop. This is a straightforward way to avoid overfitting and save computational resources.

I am using collected and historical run data to make informed decisions on when to stop the runs. This approach ensures efficient use of resources and is my attempt at focusing efforts on promising solutions.

Model Evaluation

Optimisation Strategy in Genetic Programming for Income Prediction

My GP optimization strategy involves the selection of the best individuals across different parameter setups. Here I will detail my approach to identify and track the most effective solutions during the evolutionary process.

Tracking Best Individuals

The idea is to maintain a record of the most effective GP individuals across various runs and parameter configurations.

best_individual_per_parameter_set: A list to keep track of the best individuals for each combination of crossover and mutation probabilities.

best_individual_per_guess: A list that records the best individuals from each guess within a run.

best_individual_overall: A variable to store the overall best individual across all runs and guesses.

Parameter Exploration

Crossover Probabilities (p_crossover_values): Values ranging from [0.4, 0.6, 0.7, 0.8, 0.9] are explored to understand how different rates of crossover affect the evolution of solutions.

Mutation Probabilities (p_mutation_values): Values ranging from [0.01, 0.1, 0.3] are used. The selection of this range is intended to observe the effects ranging from minimal mutation at 0.01, which promotes subtle genetic diversity, to more pronounced mutation rates 0.3, potentially leading to significant genetic variations within the population.

Evolutionary Runs

For each combination of crossover and mutation probabilities, multiple guesses (iterations) are executed. This allows the model to explore a variety of evolutionary paths, increasing the likelihood of discovering optimal solutions.

Multi-Dimensional Optimisation

By varying crossover and mutation rates, the strategy looks at the influence of genetic diversity on the evolution of solutions.The chosen range of crossover and mutation probabilities ensures a comprehensive exploration from conservative to aggressive genetic mixing.

Best Individual Selection

The process of recording the best individuals for each parameter set and each guess ensures that no potential optimal solution is overlooked.The comparison of individuals across all runs and parameters to find the best overall individual is to try to ensure the highest quality of the final model.

Visualisation and Analysis

I am plotting fitness values across generations for each parameter set that provides visual insights into the evolutionary dynamics to help aid in the analysis and selection of the best individuals.

Results

I ran the model using multiple different configurations, below are all of the individuals over time.

Unfortunately, overfitting was a big issue and due to time constraints, I did not get time to address this.

All combinations of mutation and crossover were tested in each setup

Crossover: [0.4, 0.6]

Mutation: [0.01, 0.1, 0.3]

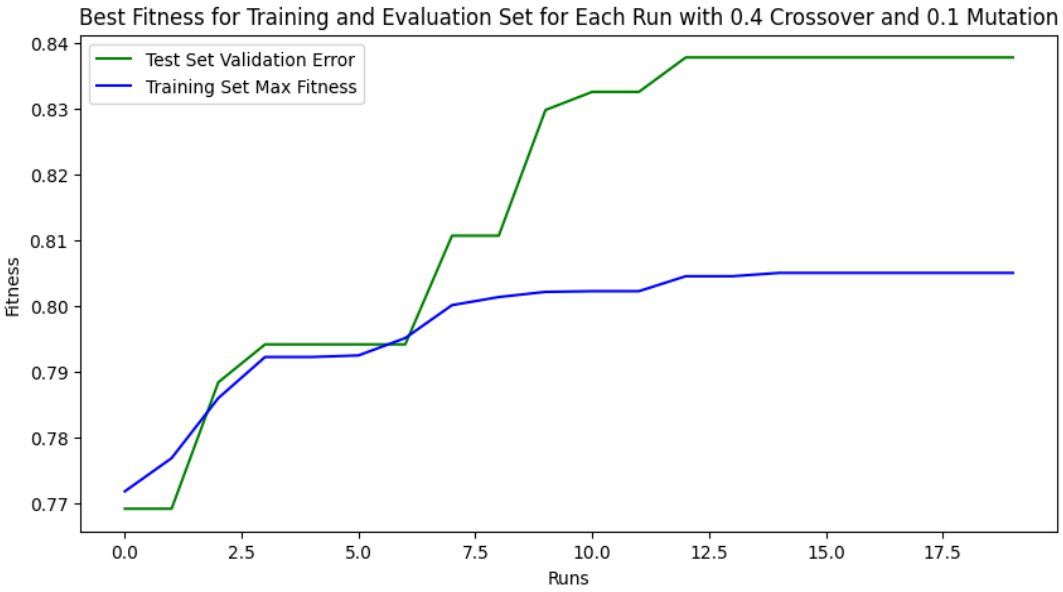
Setup 1 (Encoder: catboost, Train:Test Split: 80:20):

```
5 solutions fit our parameters

Best solution based on validation error:
f=0.8050997222923504_v=0.8378661087866108

Best solution found with configuration: P_CROSSOVER: 0.4 and P_MUTATION:
0.1
```

P_CROSSOVER: 0.4 and P_MUTATION: 0.1

Guess 1

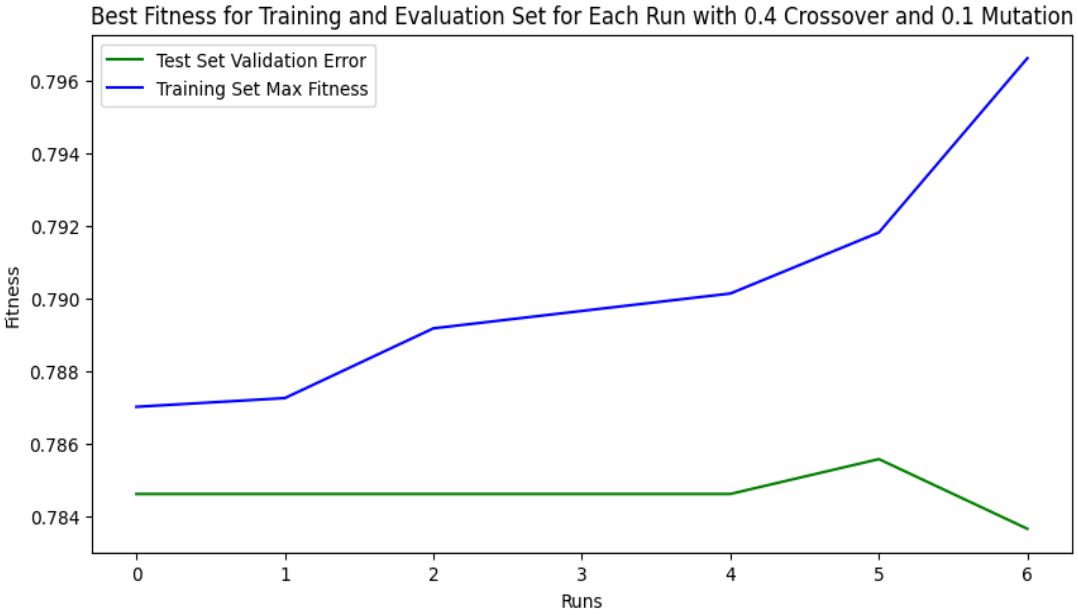
Result: f=0.8050997222923504_v=0.8378661087866108



P_CROSSOVER: 0.4 and P_MUTATION: 0.1

Guess 2

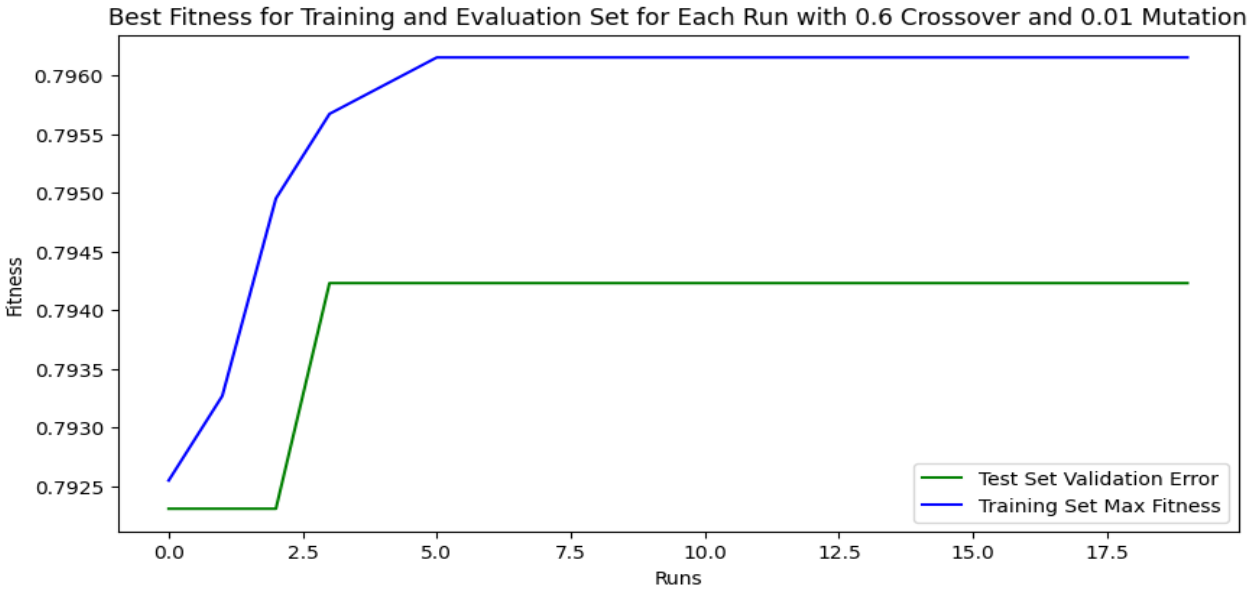
Result: f=0.7966346153846153_v=0.7836538461538461



P_CROSSOVER: 0.6 and P_MUTATION: 0.01

Guess 2

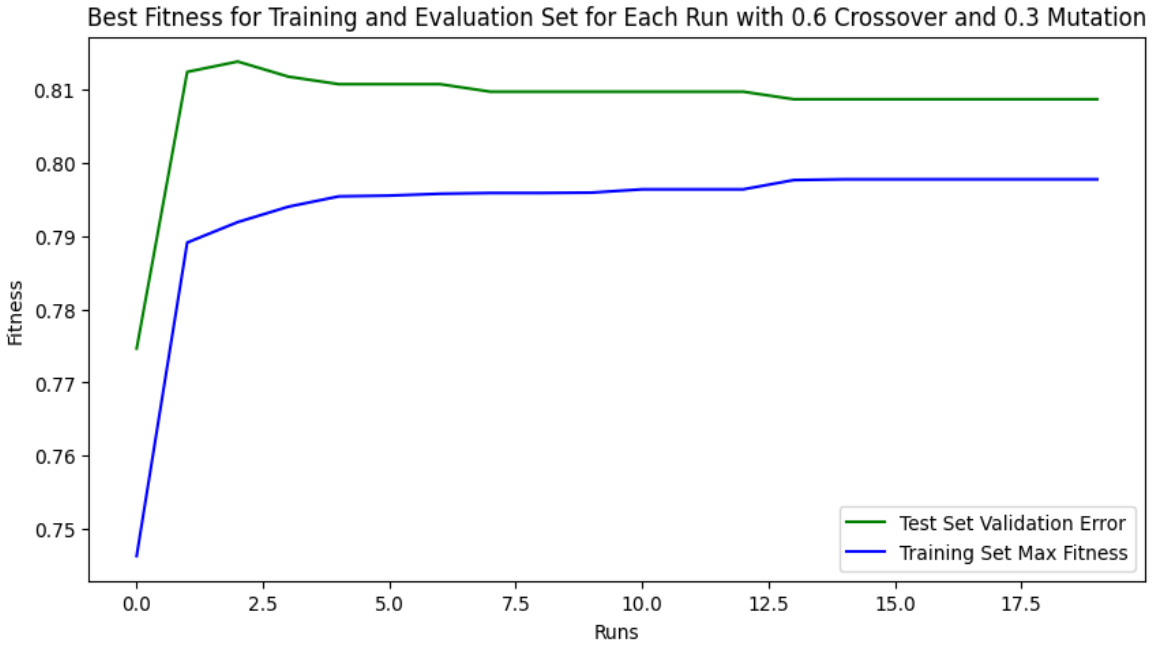
Result: f=0.7961538461538461_v=0.7942307692307692



P_CROSSOVER: 0.6 and P_MUTATION: 0.3

Guess 2

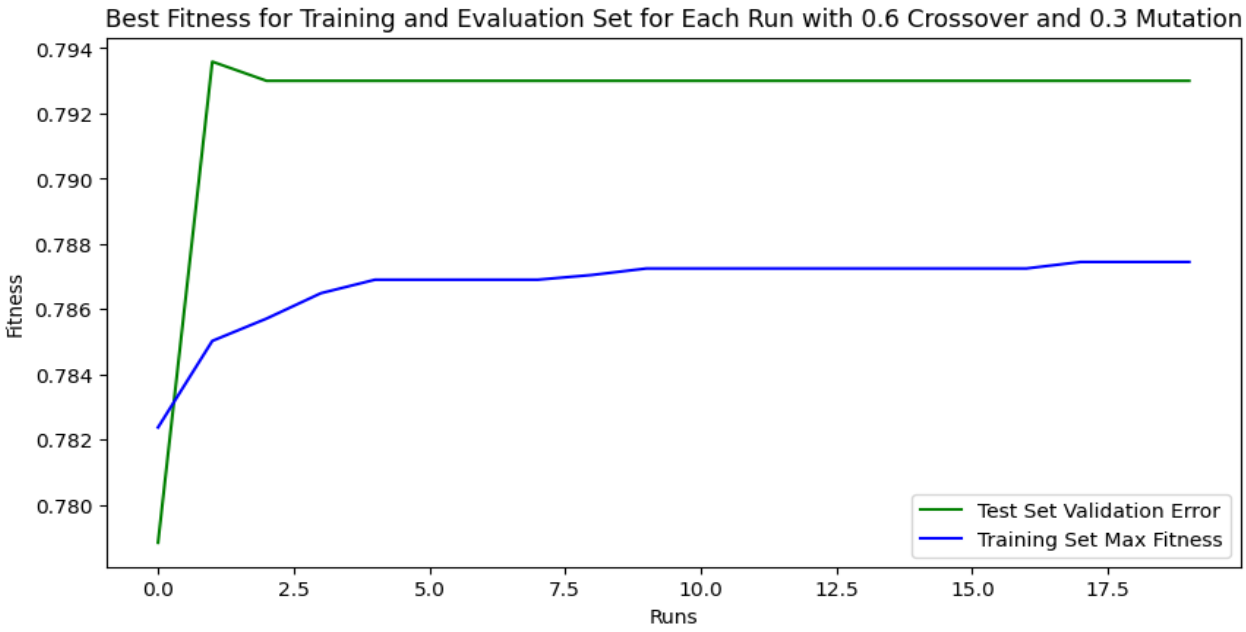
Result: f=0.797724399494311_v=0.808641975308642



P_CROSSOVER: 0.6 and P_MUTATION: 0.3

Guess 1

Result: f=0.7874493927125507_v=0.793



Setup 2 (Encoder: target, Train:Test Split: 80:20):

11 solutions fit our parameters

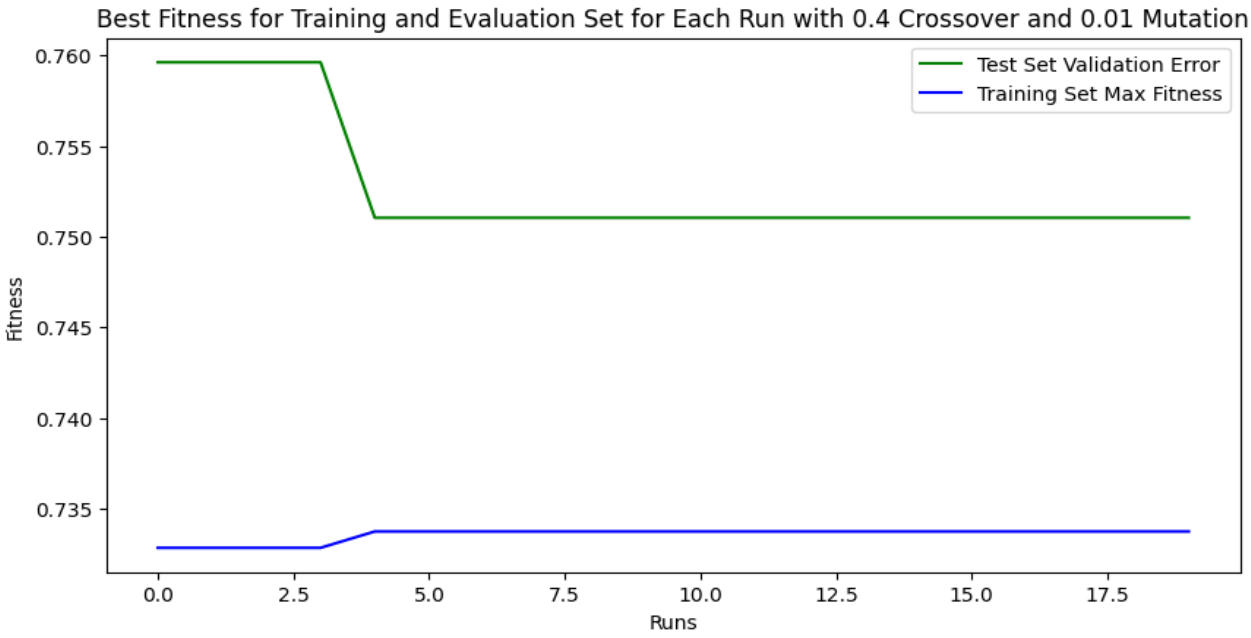
Best solution based on validation error:
f=0.8016696180116367_v=0.8525564803804994

Best solution found with configuration: P_CROSSOVER: 0.4 and P_MUTATION: 0.1

P_CROSSOVER: 0.4 and P_MUTATION: 0.01

Guess 2

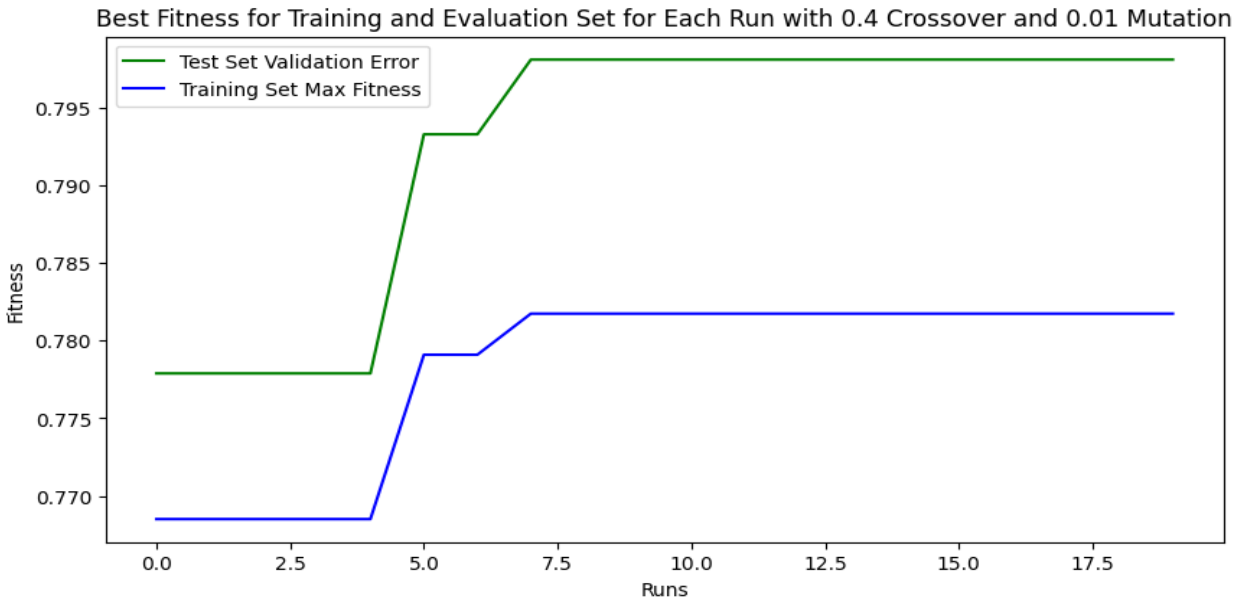
Result: f=0.7337322363500374_v=0.7510373443983402



P_CROSSOVER: 0.4 and P_MUTATION: 0.01

Guess 3

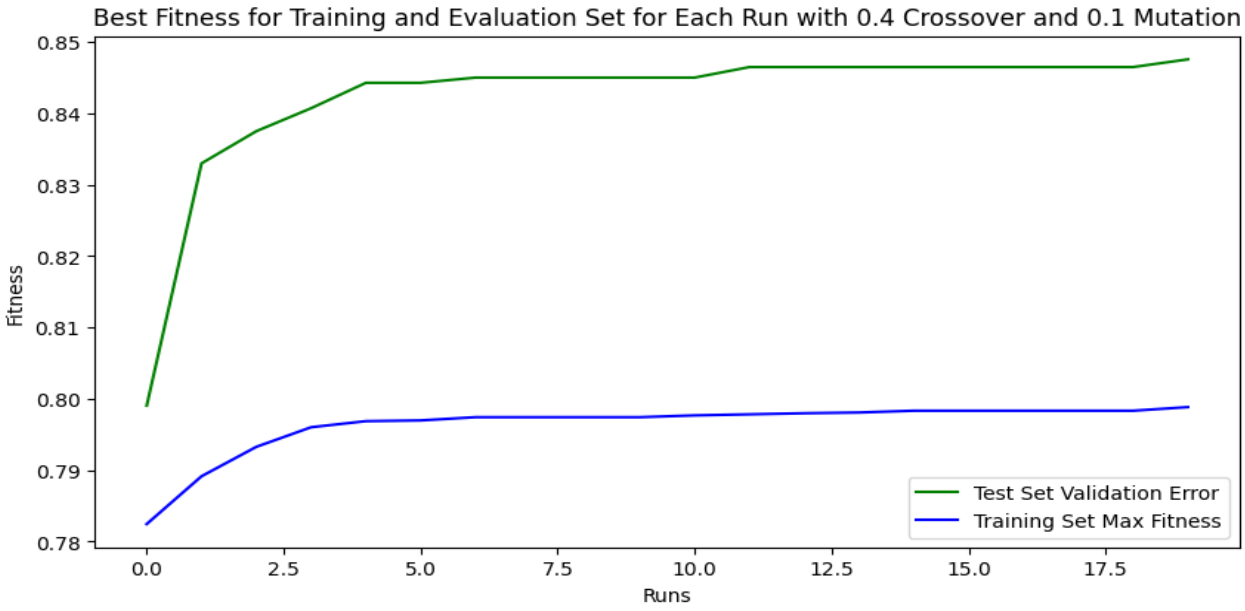
Result: f=0.7817307692307692_v=0.7980769230769231



P_CROSSOVER: 0.4 and P_MUTATION: 0.1

Guess 1

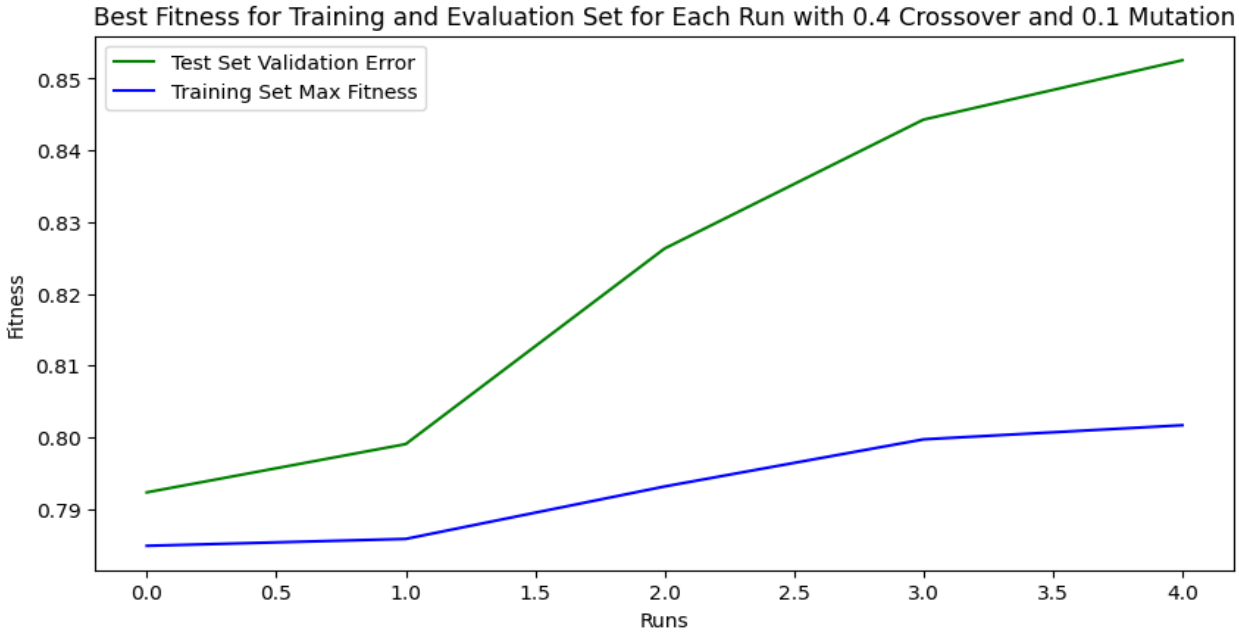
Result: f=0.798836032388664_v=0.8475675675675676



P_CROSSOVER: 0.4 and P_MUTATION: 0.1

Guess 2

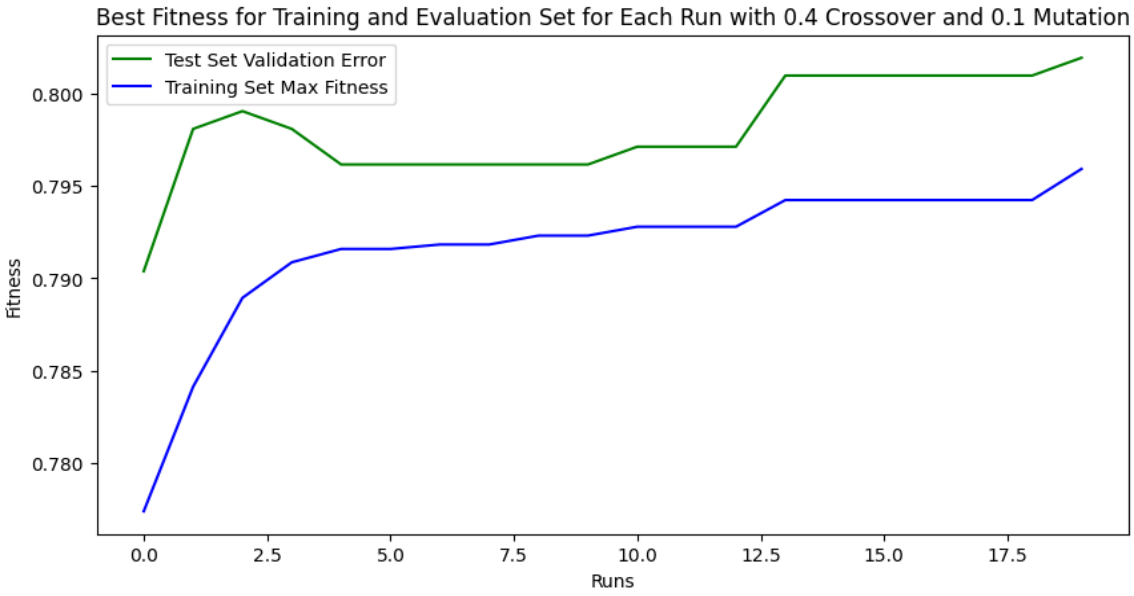
Result: f=0.8016696180116367_v=0.8525564803804994



P_CROSSOVER: 0.4 and P_MUTATION: 0.1

Guess 3

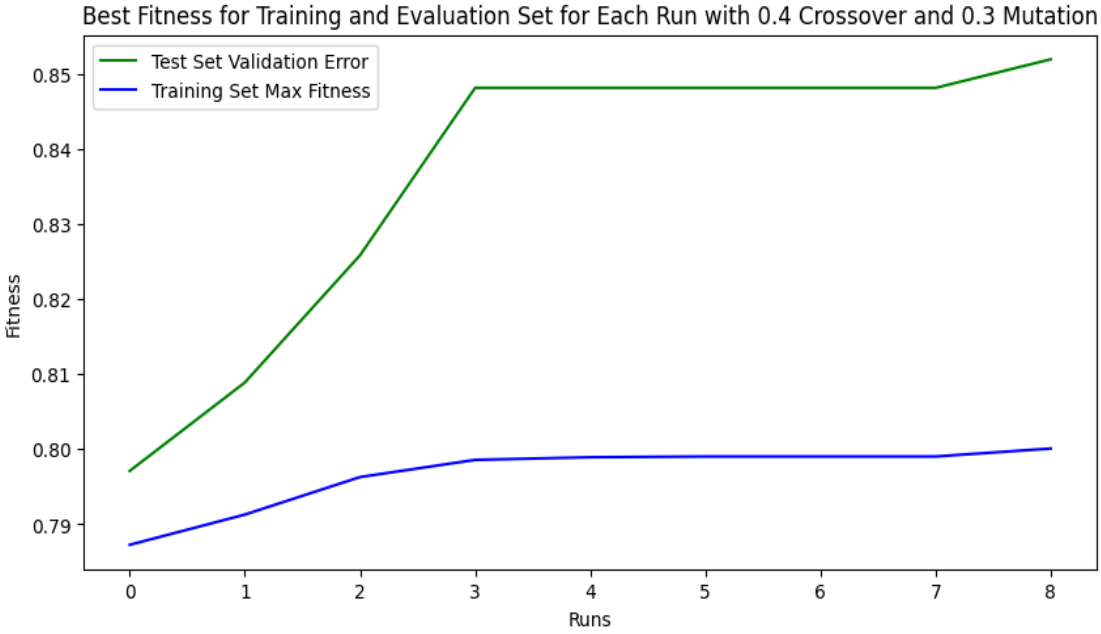
Result: f=0.7959134615384615_v=0.801923076923077



P_CROSSOVER: 0.4 and P_MUTATION: 0.3

Guess 1

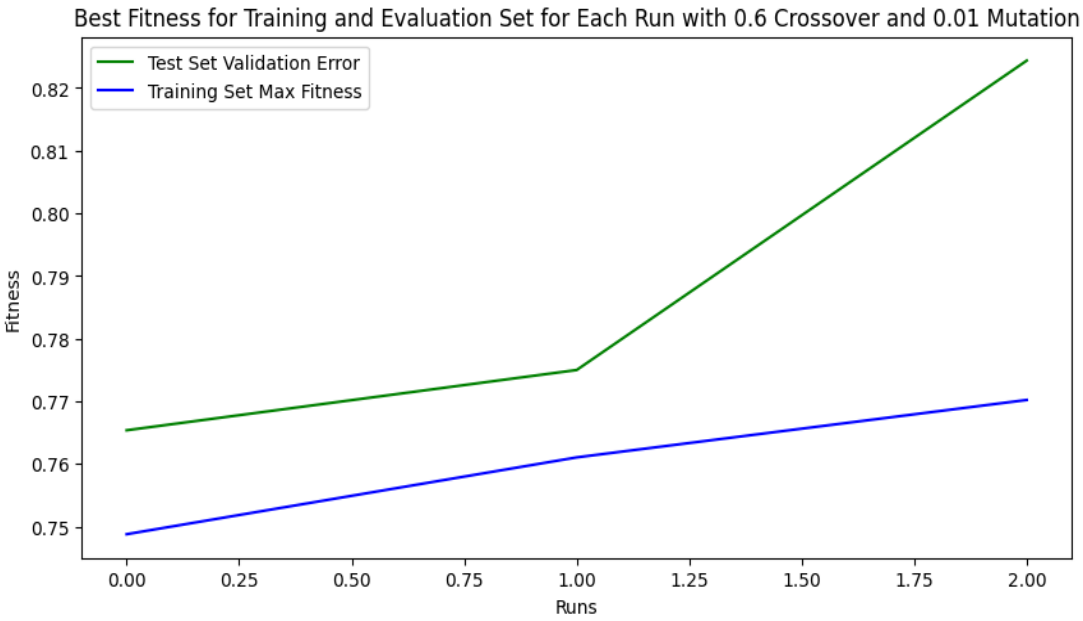
Result: f=0.8001010866818297_v=0.8520084566596194



P_CROSSOVER: 0.6 and P_MUTATION: 0.01

Guess 3

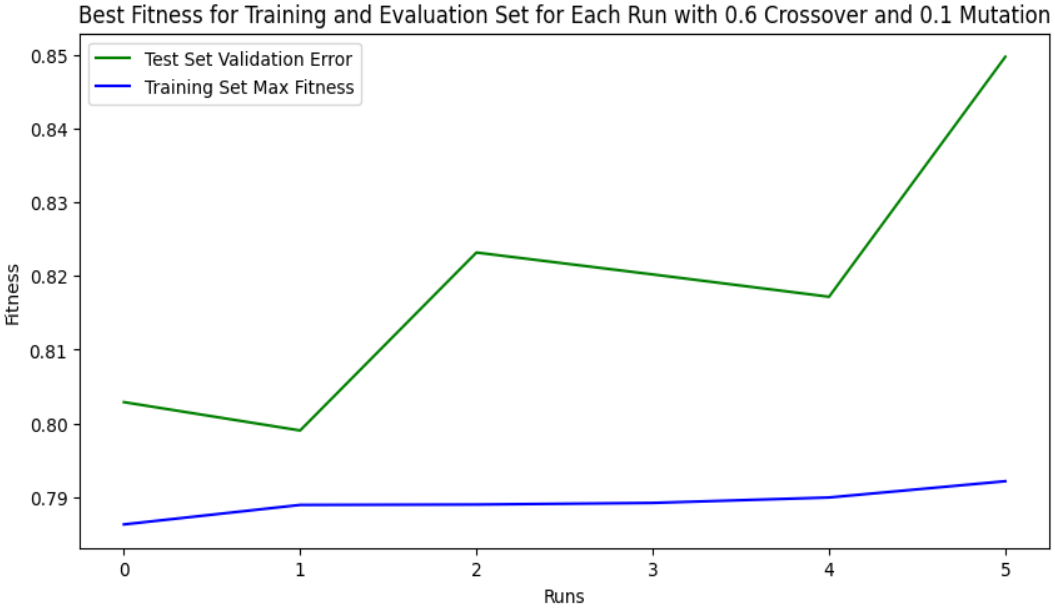
Result: f=0.7702224469160769_v=0.8243688254665203



P_CROSSOVER: 0.6 and P_MUTATION: 0.1

Guess 2

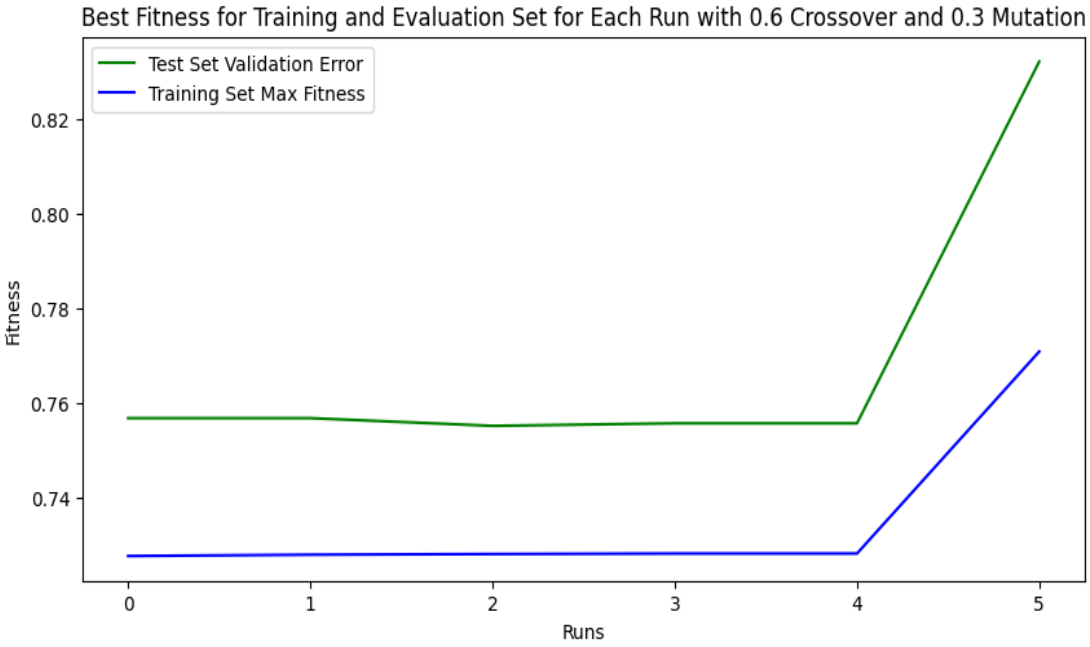
Result: f=0.7921588338778588_v=0.8497175141242937



P_CROSSOVER: 0.6 and P_MUTATION: 0.3

Guess 1

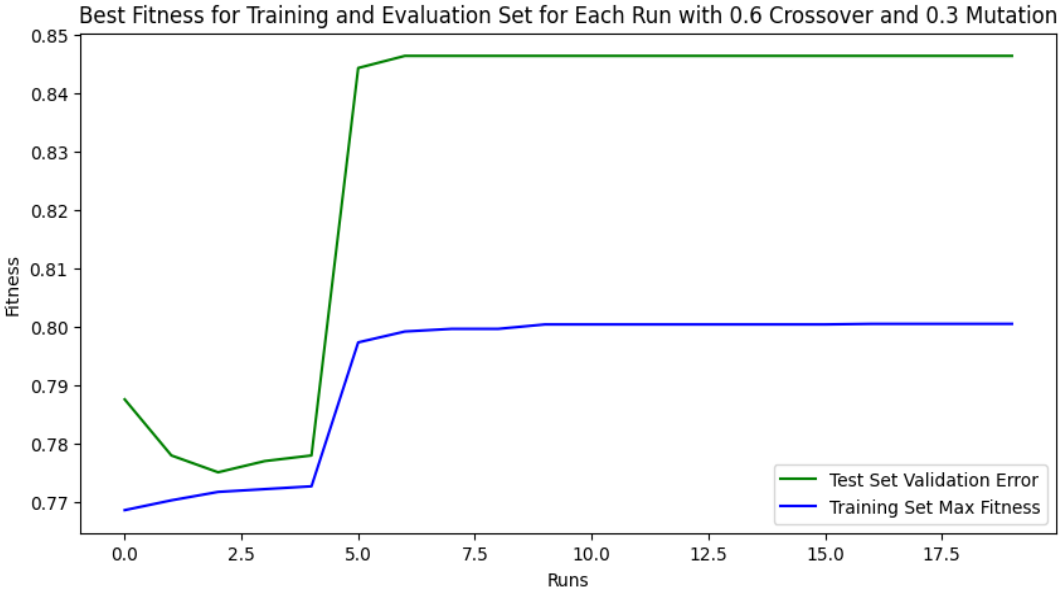
Result: f=0.771002024291498_v=0.832271762208068



P_CROSSOVER: 0.6 and P_MUTATION: 0.3

Guess 2

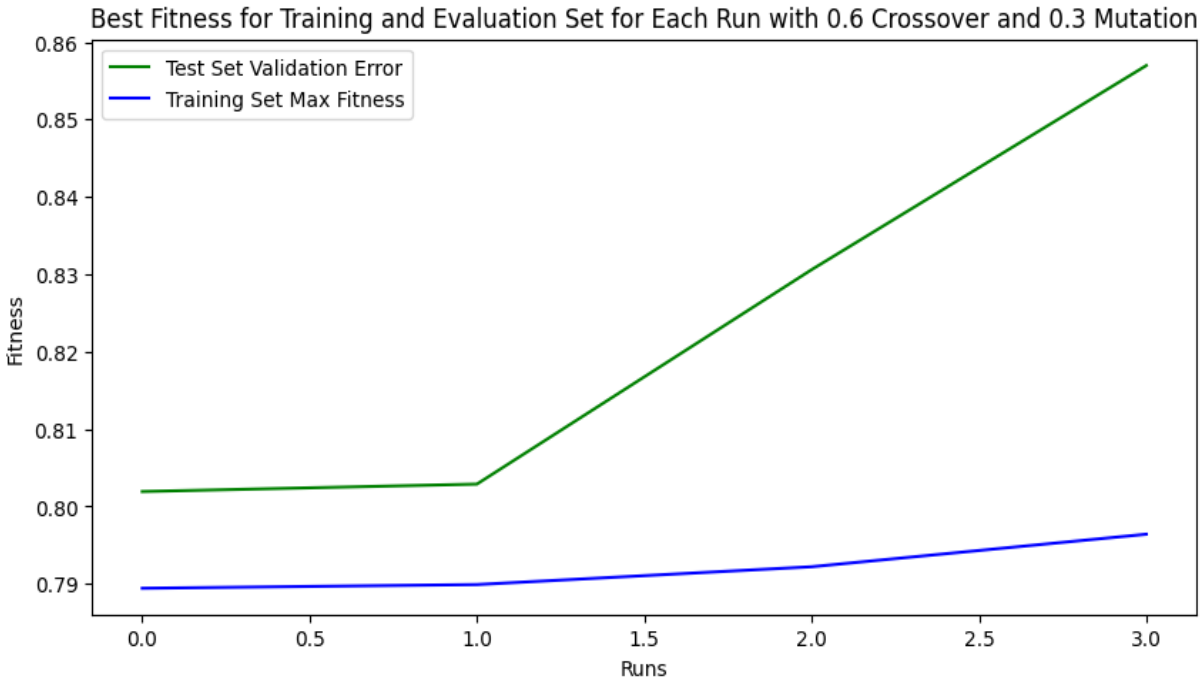
Result: f=0.8004552352048558_v=0.8464017185821697



P_CROSSOVER: 0.6 and P_MUTATION: 0.3

Guess 3

Result: f=0.7964087000505817_v=0.8569780853517878



Setup 3 (Encoder: catboost, Train:Test Split: 70:30):

8 solutions fit our parameters

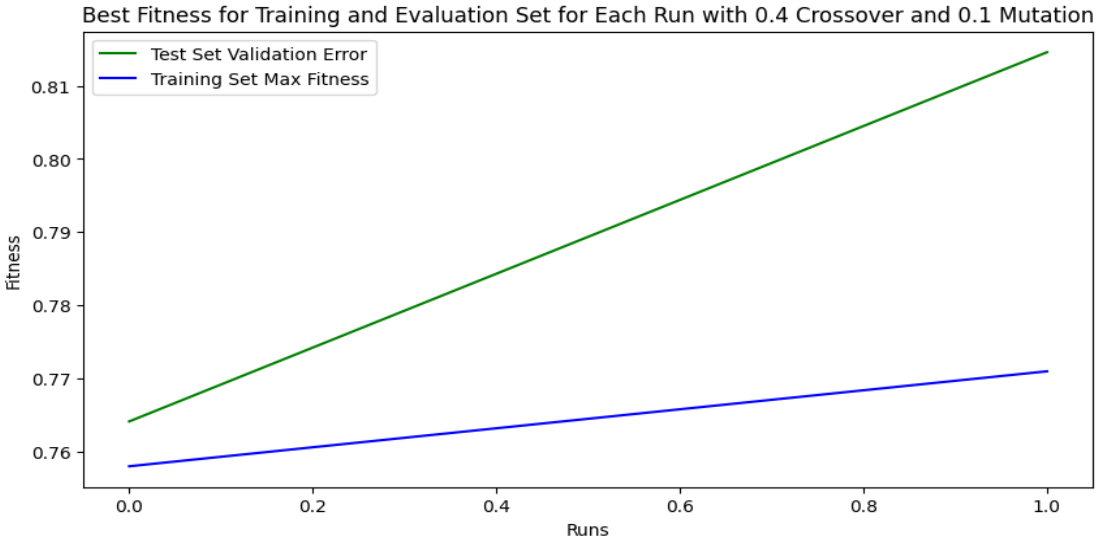
Best solution based on validation error:
f=0.7935889754343919_v=0.844776119402985

Best solution found with configuration: P_CROSSOVER: 0.6 and P_MUTATION: 0.1

P_CROSSOVER: 0.4 and P_MUTATION: 0.1

Guess 1

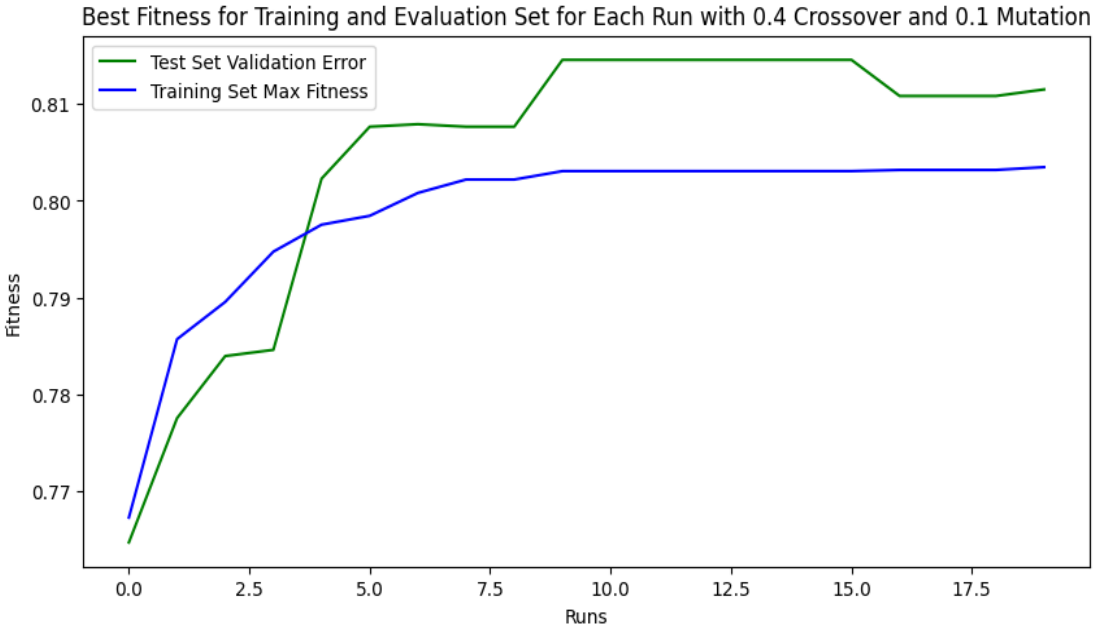
Result: f=0.7709593777009507_v=0.8146067415730337



P_CROSSOVER: 0.4 and P_MUTATION: 0.1

Guess 3

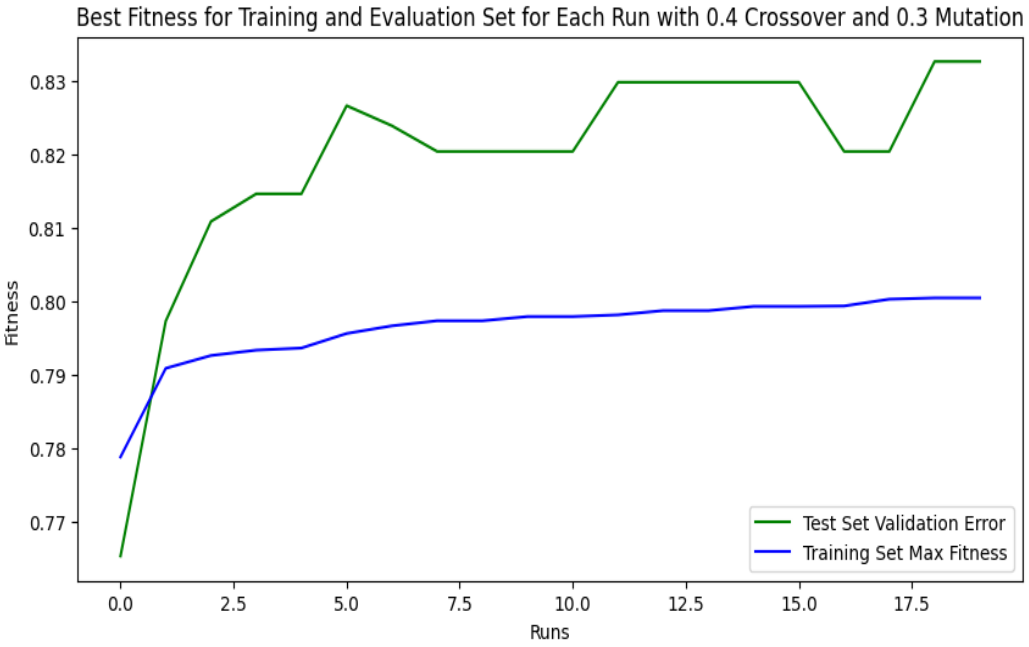
Result: f=0.8034682080924855_v=0.8114864864864865



P_CROSSOVER: 0.4 and P_MUTATION: 0.3

Guess 1

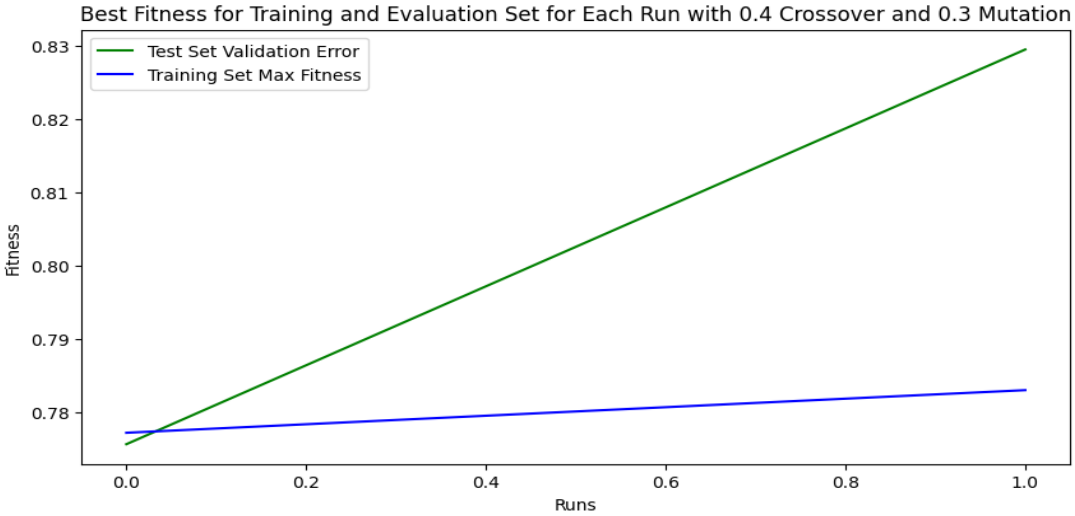
Result: f=0.8004620271440948_v=0.8325872873769025



P_CROSSOVER: 0.4 and P_MUTATION: 0.3

Guess 2

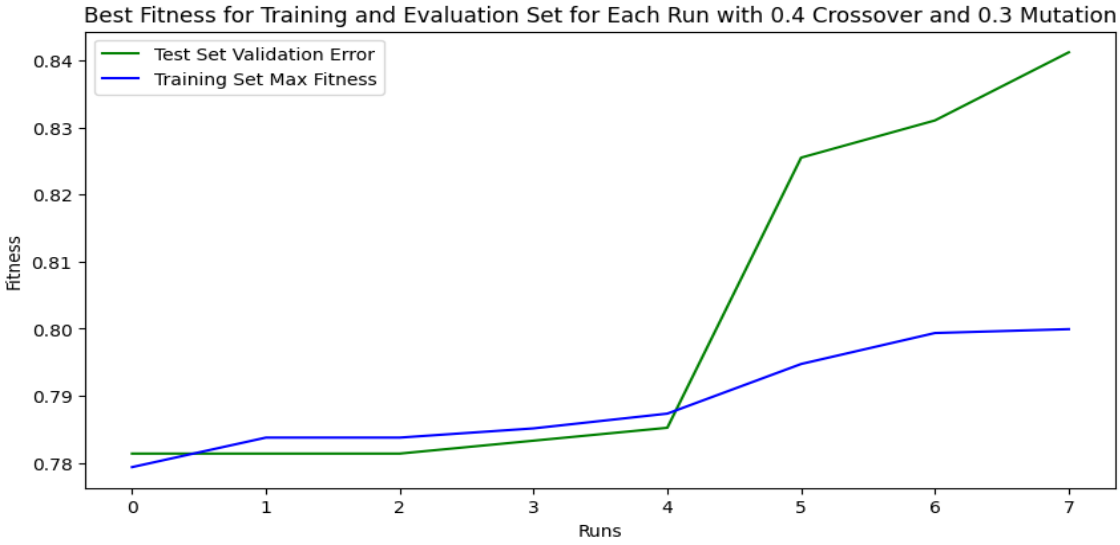
Result: f=0.78300803673938_v=0.8294515401953418



P_CROSSOVER: 0.4 and P_MUTATION: 0.3

Guess 3

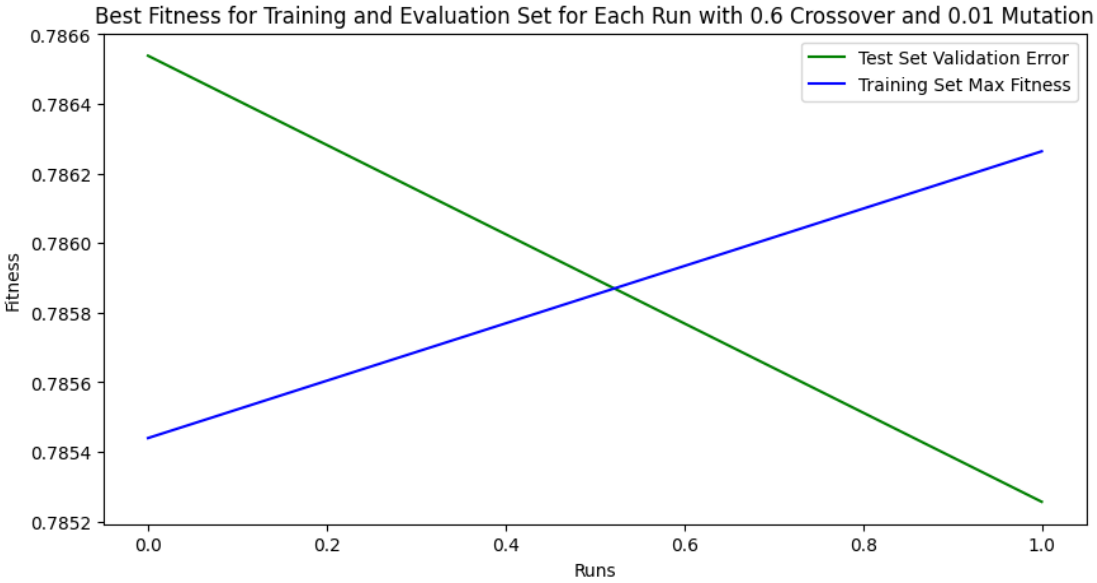
Result: f=0.7999421798207574_v=0.8411680911680912



P_CROSSOVER: 0.6 and P_MUTATION: 0.01

Guess 3

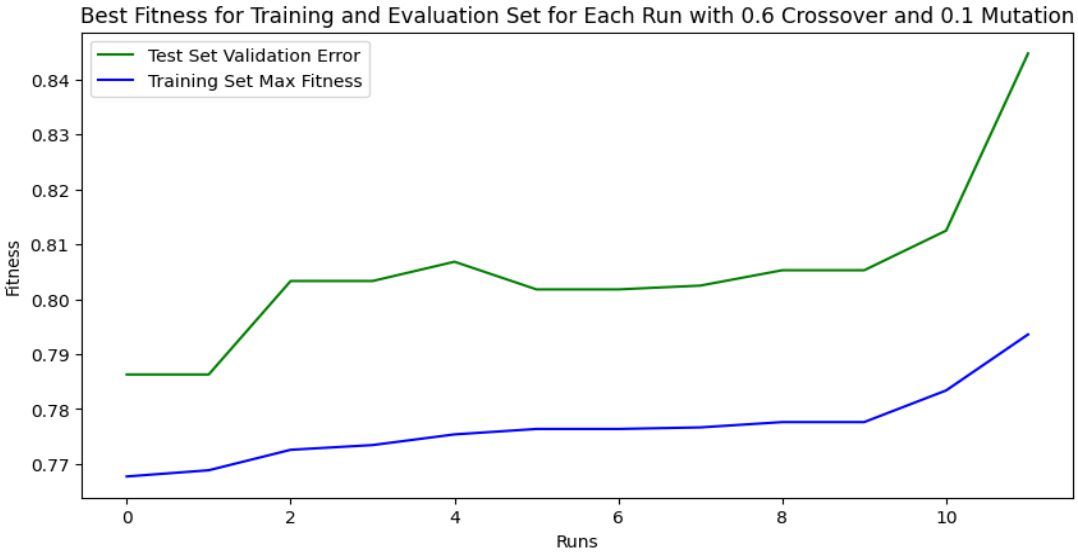
Result: f=0.7862637362637362_v=0.7852564102564102



P_CROSSOVER: 0.6 and P_MUTATION: 0.1

Guess 1

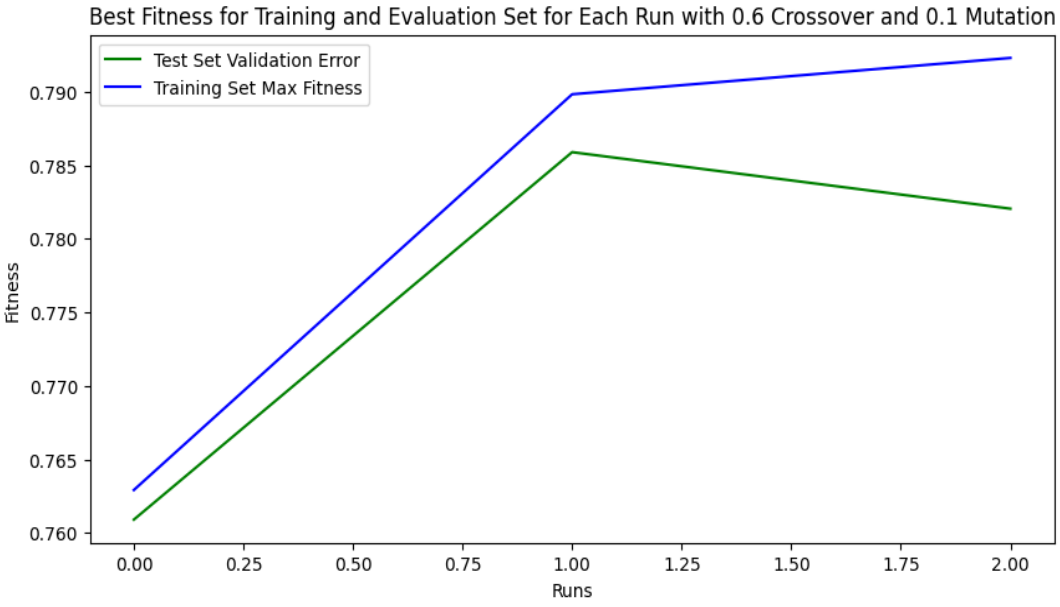
Result: f=0.7935889754343919_v=0.844776119402985



P_CROSSOVER: 0.6 and P_MUTATION: 0.1

Guess 2

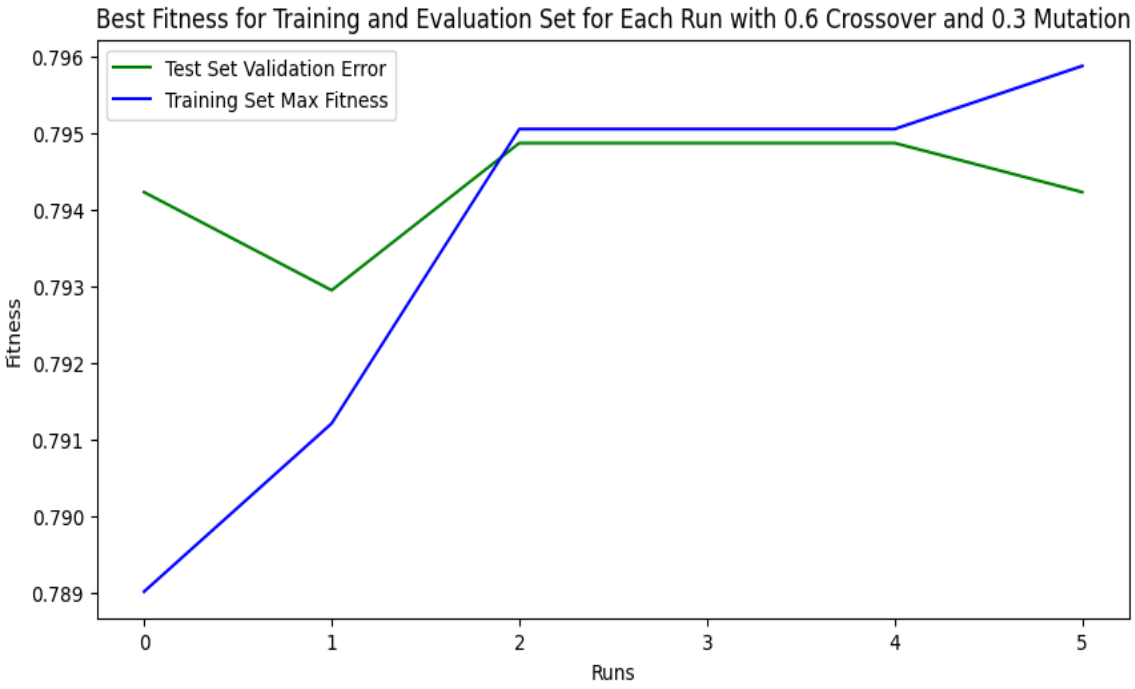
Result: f=0.7923076923076923_v=0.782051282051282



P_CROSSOVER: 0.6 and P_MUTATION: 0.3

Guess 3

Result: f=0.7958791208791208_v=0.7942307692307692



Setup 4 (Encoder: target, Train:Test Split: 70:30):

6 solutions fit our parameters

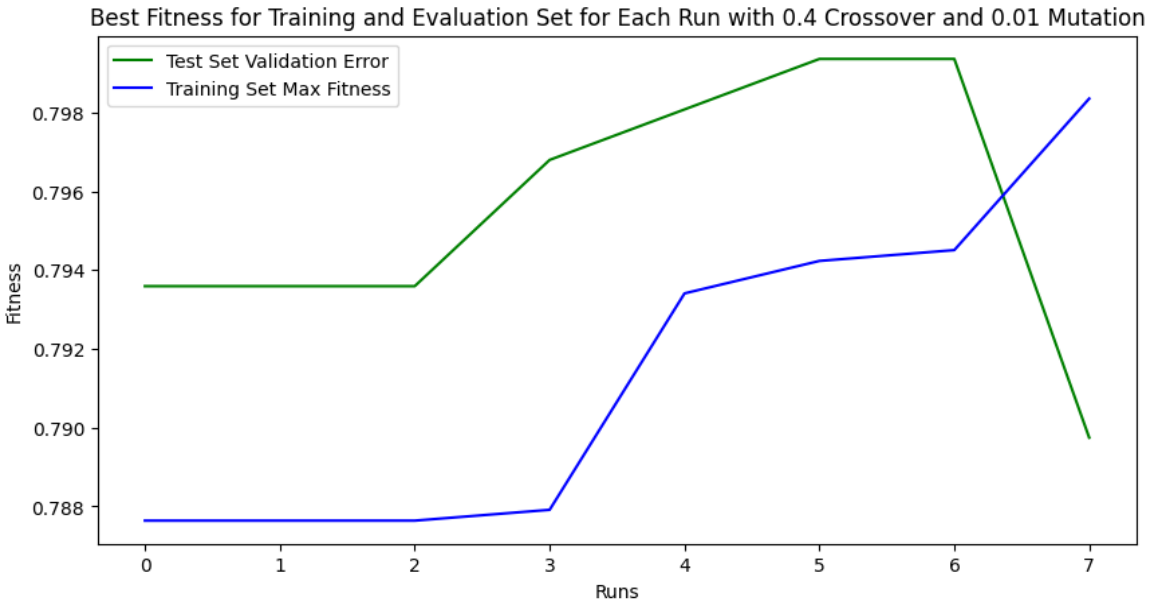
Best solution based on validation error:
f=0.8119040739670615_v=0.8514025777103866

Best solution found with configuration: P_CROSSOVER: 0.6 and P_MUTATION: 0.3

P_CROSSOVER: 0.4 and P_MUTATION: 0.01

Guess 2

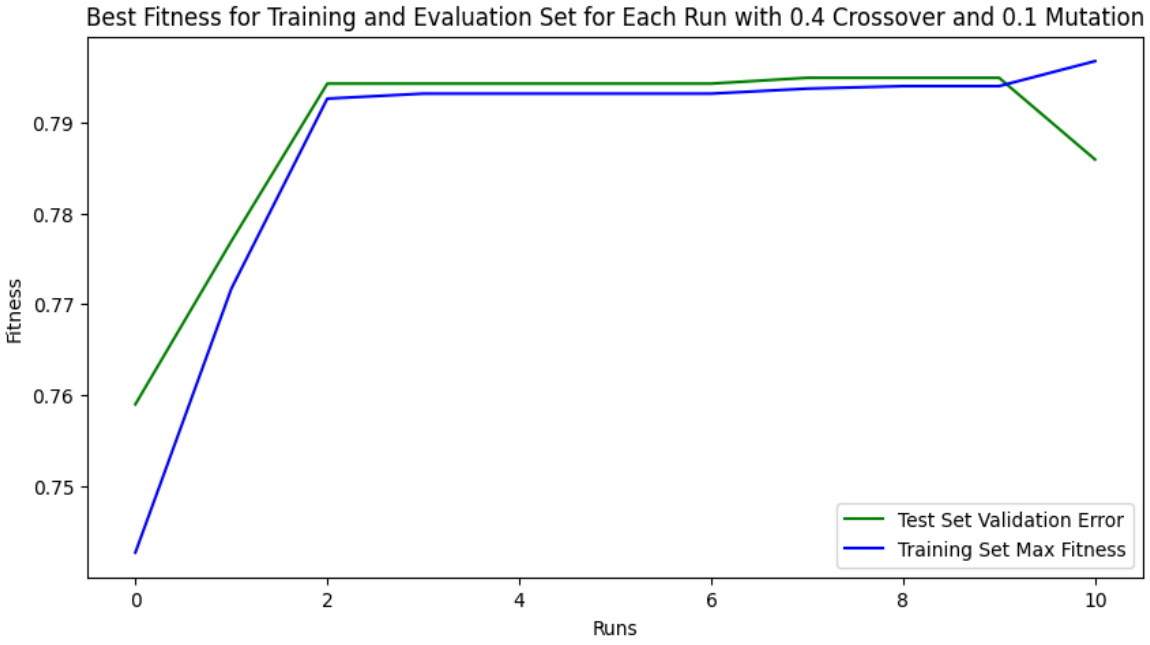
Result: f=0.7945054945054945_v=0.7993589743589744



P_CROSSOVER: 0.4 and P_MUTATION: 0.1

Guess 3

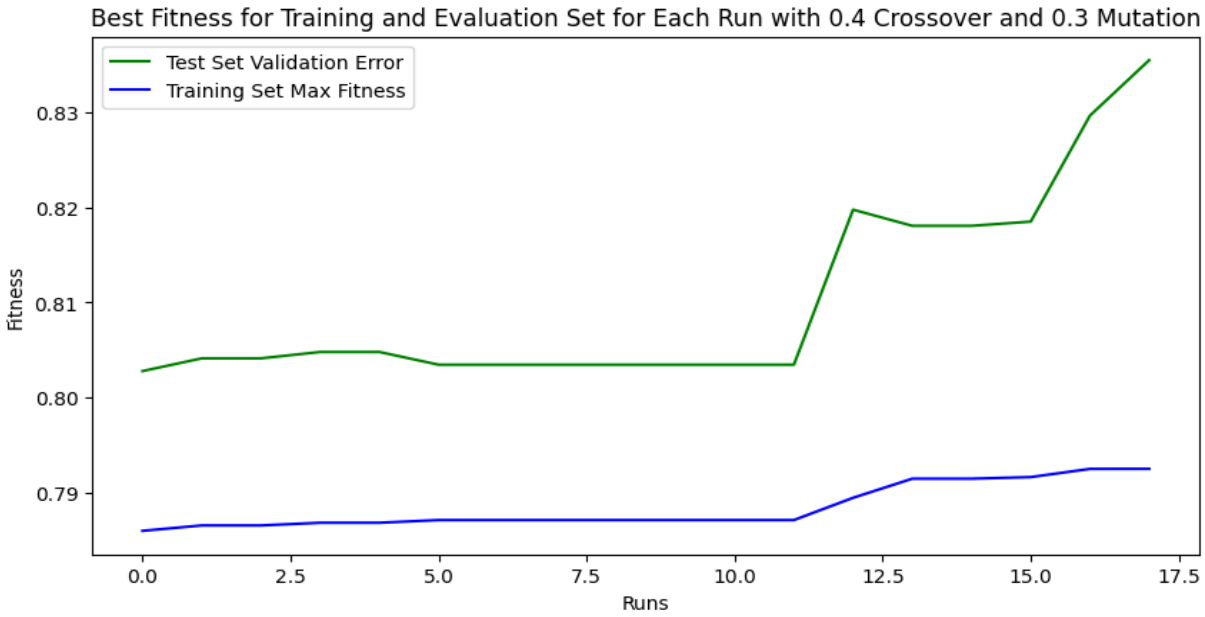
Result: f=0.7939560439560439_v=0.7948717948717948



P_CROSSOVER: 0.4 and P_MUTATION: 0.3

Guess 2

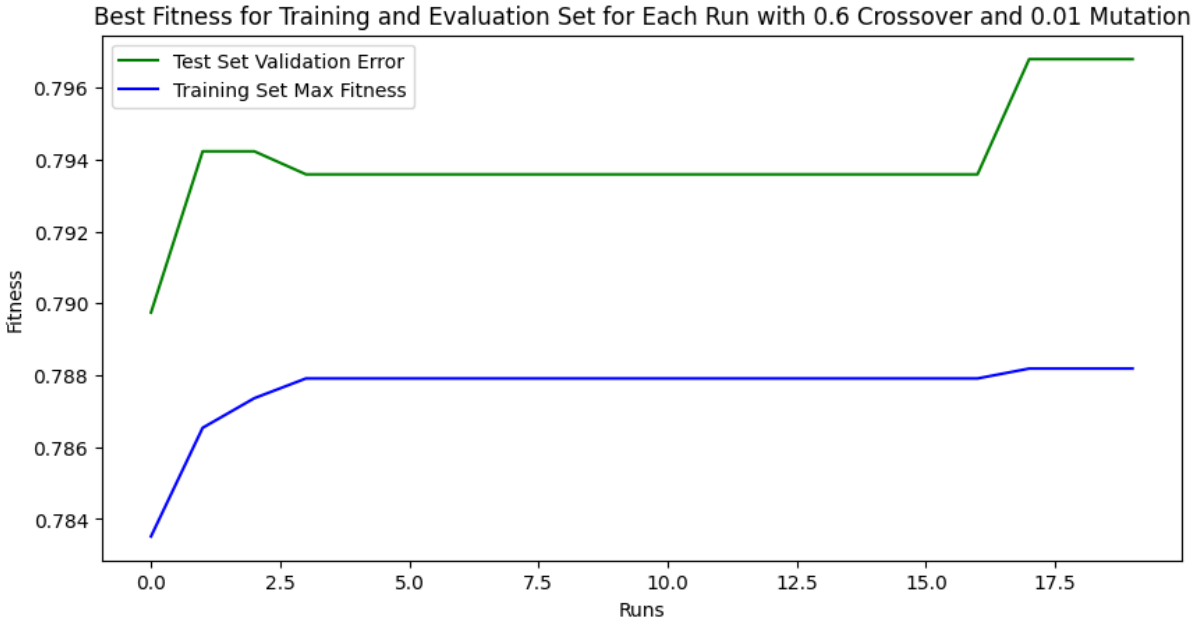
Result: f=0.7925455070788789_v=0.8354430379746836



P_CROSSOVER: 0.6 and P_MUTATION: 0.01

Guess 2

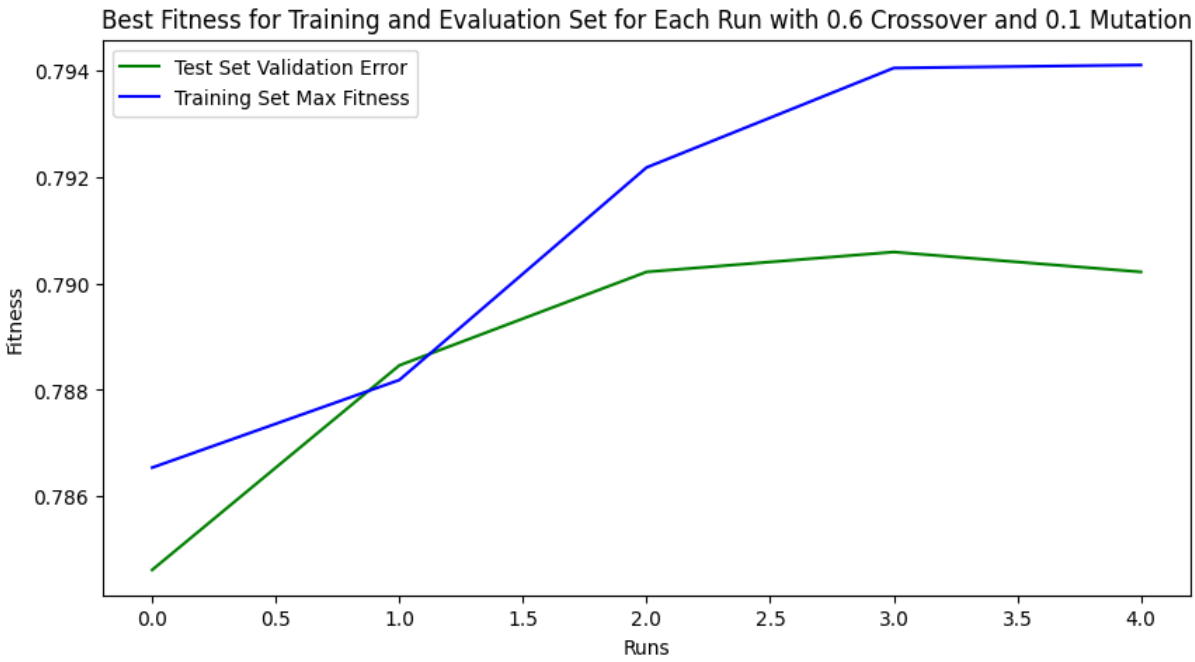
Result: f=0.7881868131868132_v=0.7967948717948717



P_CROSSOVER: 0.6 and P_MUTATION: 0.1

Guess 2

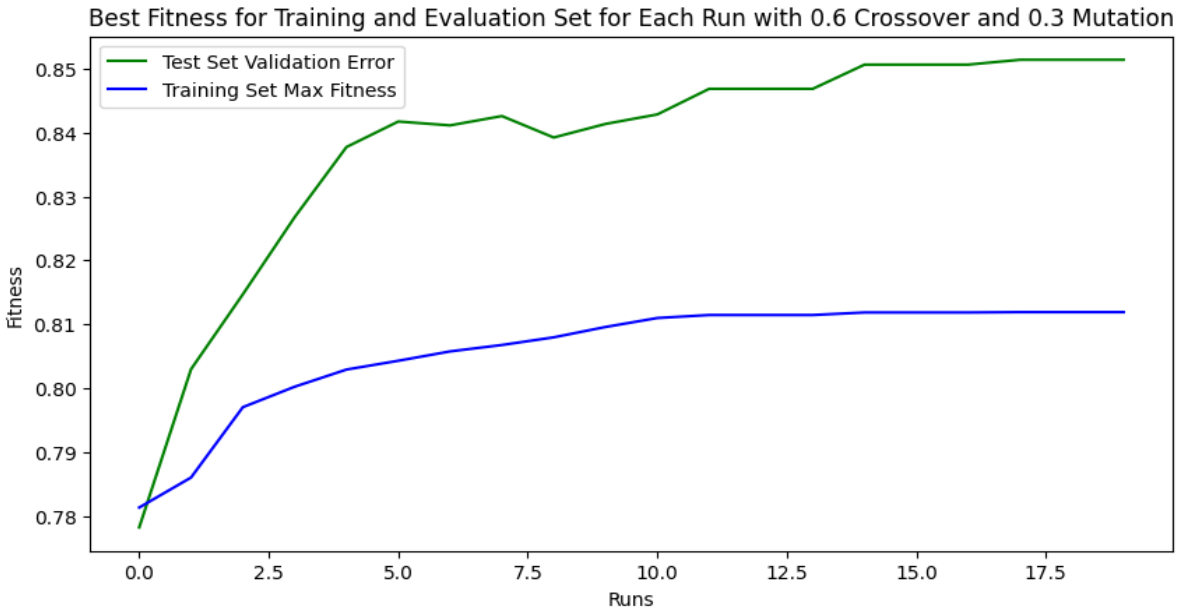
Result: f=0.7940528634361234_v=0.7905927835051546



P_CROSSOVER: 0.6 and P_MUTATION: 0.3

Guess 3

Result: f=0.8119040739670615_v=0.8514025777103866



Best Result Overall

Best solution based on validation error:
f=0.8016696180116367_v=0.8525564803804994

Currently on run 1 of 20

The execution time is 217.77748203277588 seconds.

-- Best Individual = lt(lt(ARG1, mul(add(ARG8,
sub(psin(0.6269646488263211), gt(logical_not(ARG2), add(add(ARG8, ARG2),
psin(0.6269646488263211))))), psin(add(psin(0.6269646488263211),
sub(psin(lt(ARG1, mul(lt(ARG8, sub(psin(0.6269646488263211),
gt(logical_not(ARG1), add(ARG1, ARG2)))), psin(add(ARG8,
sub(psin(0.6269646488263211), gt(ARG8, add(add(ARG8, ARG2), ARG2)))))))),
gt(logical_not(ARG1), add(add(ARG8, logical_not(ARG1)), ARG2)))))),
add(ARG8, ARG2))

Maximum fitness achieved: 0.7848557692307693

Validation Accuracy: 0.7923076923076923

CSV file saved to: F:/Nextcloud/University/Sem1/CS6271 EVOLUTIONARY
COMPUTATION AND HUMANOID
ROBOTICS/kaggle_submissions/kaggle_submission__f=0.7848557692307693_v=0.79
23076923076923.csv

Currently on run 2 of 20

The execution time is 275.9601638317108 seconds.

-- Best Individual = lt(lt(ARG1, mul(add(ARG8,
sub(psin(0.6269646488263211), add(for_loop(ARG1, ARG4, ARG3),
neg(0.574175486624329)))), psin(add(psin(0.6269646488263211),
sub(psin(lt(ARG1, mul(lt(ARG8, sub(psin(0.6269646488263211),
gt(logical_not(ARG1), add(ARG1, ARG2)))), psin(add(ARG8,
sub(psin(0.6269646488263211), gt(ARG8, add(add(ARG8, ARG2), ARG2)))))))),
gt(logical_not(ARG1), add(add(ARG8, add(ARG8,
sub(psin(0.6269646488263211), add(for_loop(ARG1, ARG4, ARG3),
neg(0.574175486624329))))), ARG2)))))), add(ARG8, ARG2))

Maximum fitness achieved: 0.7858173076923077

Validation Accuracy: 0.7990384615384616

CSV file saved to: F:/Nextcloud/University/Sem1/CS6271 EVOLUTIONARY COMPUTATION AND HUMANOID ROBOTICS/kaggle_submissions/kaggle_submission__f=0.7858173076923077_v=0.7990384615384616.csv

Currently on run 3 of 20

The execution time is 290.2317464351654 seconds.

-- Best Individual = lt(lt(ARG1, mul(add(ARG8, sub(psin(0.6269646488263211), add(for_loop(logical_not(0.574175486624329), ARG4, ARG3), neg(0.574175486624329)))), add(mul(add(ARG8, sub(psin(0.6269646488263211), add(for_loop(add(ARG8, sub(psin(0.6269646488263211), ARG2)), ARG4, add(ARG4, neg(0.574175486624329))), neg(0.685175221795631)))), add(psin(sub(add(ARG8, add(ARG8, sub(psin(0.6269646488263211), add(for_loop(ARG4, ARG4, mul(ARG6, ARG2)), neg(0.574175486624329))))), gt(logical_not(ARG1), add(ARG1, ARG2)))), sub(psin(lt(ARG3, ARG4)), gt(ARG2, 0.574175486624329)))), sub(psin(lt(ARG1, ARG4)), gt(ARG2, add(add(ARG8, add(ARG8, sub(psin(0.6269646488263211), add(for_loop(ARG1, ARG4, ARG3), neg(0.574175486624329))))), ARG2)))))), add(ARG8, ARG2))

Maximum fitness achieved: 0.7931288343558283

Validation Accuracy: 0.8263052208835341

CSV file saved to: F:/Nextcloud/University/Sem1/CS6271 EVOLUTIONARY COMPUTATION AND HUMANOID ROBOTICS/kaggle_submissions/kaggle_submission__f=0.7931288343558283_v=0.8263052208835341.csv

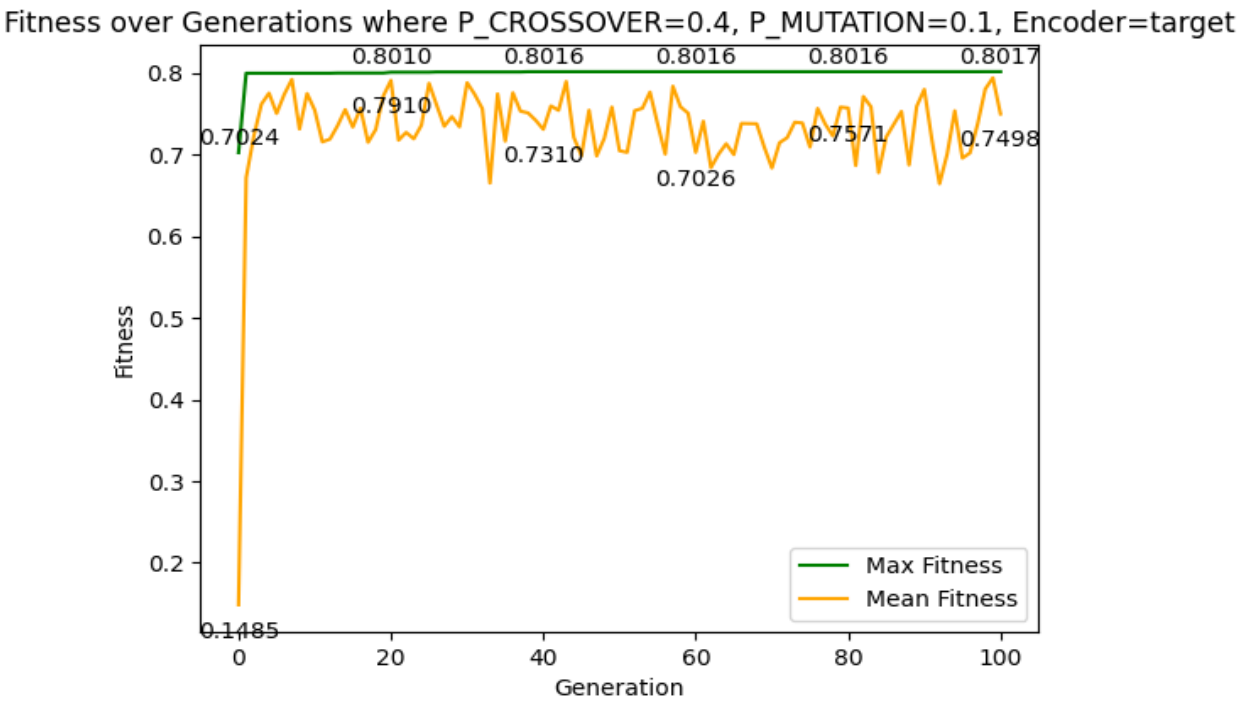
Currently on run 4 of 20

The execution time is 317.93616819381714 seconds.

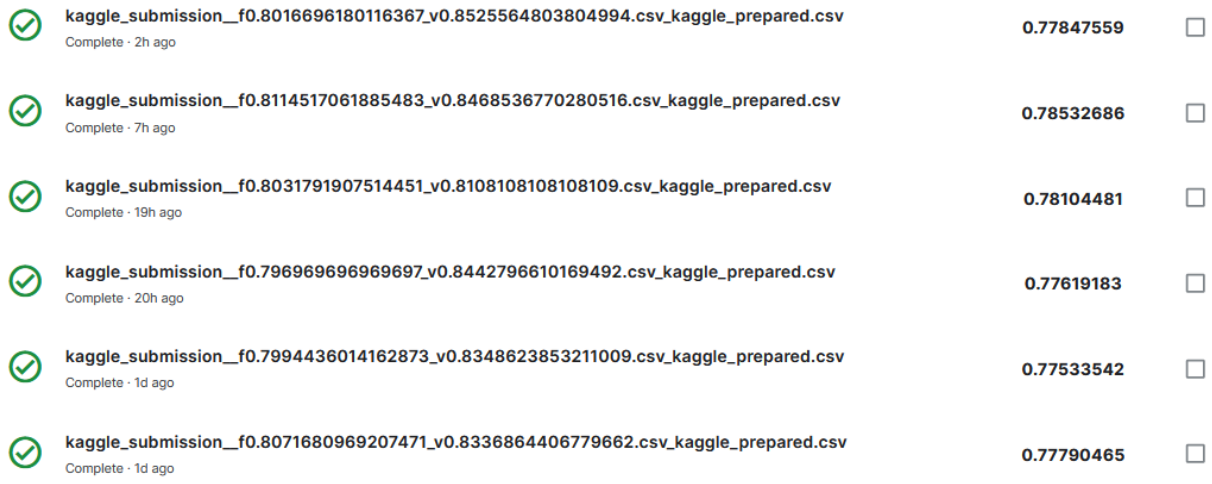
-- Best Individual = lt(lt(ARG1, mul(0.685175221795631, add(mul(add(ARG8, sub(psin(0.6269646488263211), add(for_loop(logical_or(ARG4, ARG6), ARG7, neg(sub(ARG8, mul(ARG1, neg(sub(ARG8, mul(ARG1, ARG6)))))))), neg(psin(0.6269646488263211))))), add(psin(sub(add(ARG8, add(ARG8, sub(psin(0.6269646488263211), add(for_loop(sub(0.574175486624329, add(for_loop(ARG1, ARG4, ARG3), neg(0.574175486624329))), ARG4, mul(ARG9,

```
neg(sub(ARG8, ARG4)))), neg(0.574175486624329))))), psin(lt(ARG1,
ARG4)))), sub(psin(lt(ARG3, ARG4)), logical_and(logical_and(for_loop(ARG1,
ARG4, mul(ARG6, neg(sub(ARG8, ARG3)))), ARG8), mul(ARG1, ARG6))))),
sub(psin(lt(ARG1, ARG4)), gt(ARG2, add(add(ARG8, add(ARG8,
sub(psin(0.6269646488263211), add(for_loop(ARG1, ARG4, ARG3),
neg(0.574175486624329))))), ARG2))))))), add(ARG8, ARG2))

Maximum fitness achieved:  0.7996987951807228

Validation Accuracy:       0.8442796610169492

CSV file saved to: F:/Nextcloud/University/Sem1/CS6271 EVOLUTIONARY
COMPUTATION AND HUMANOID
ROBOTICS/kaggle_submissions/kaggle_submission__f=0.7996987951807228_v=0.84
42796610169492.csv




Currently on run 5 of 20

The execution time is 335.8651473522186 seconds.

-- Best Individual =  lt(lt(ARG1, mul(0.685175221795631, add(mul(add(ARG8,
sub(psin(0.6269646488263211), add(for_loop(logical_or(ARG4, ARG6), ARG7,
neg(sub(ARG8, mul(mul(mul(ARG1, ARG9), neg(sub(add(for_loop(ARG1, ARG3,
ARG3), neg(0.574175486624329)), mul(ARG1, ARG9)))),
neg(sub(add(for_loop(neg(0.574175486624329), ARG3, ARG3),
neg(0.574175486624329)), mul(ARG1, ARG9)))))))),
neg(psin(0.6269646488263211))))), add(psin(sub(add(ARG8, add(ARG8,
sub(psin(0.6269646488263211), add(for_loop(ARG6, ARG4, mul(ARG9,
sub(psin(ARG7), for_loop(ARG2, ARG8, 0.40601641570672653)))),
neg(0.574175486624329))))), psin(lt(ARG1, ARG4)))), sub(psin(lt(ARG3,
ARG4)), logical_and(logical_and(for_loop(0.574175486624329, ARG4,
mul(ARG6, neg(sub(ARG8, ARG3)))), ARG8), mul(ARG1, ARG6))))),
sub(psin(lt(ARG1, ARG4)), gt(ARG2, add(add(ARG8, add(ARG8,
sub(psin(0.6269646488263211), add(for_loop(ARG1, ARG4, ARG3),
neg(0.574175486624329))))), ARG2))))))), add(ARG8, ARG2))

Maximum fitness achieved:  0.8016696180116367

Validation Accuracy:       0.8525564803804994

CSV file saved to: F:/Nextcloud/University/Sem1/CS6271 EVOLUTIONARY
COMPUTATION AND HUMANOID
ROBOTICS/kaggle_submissions/kaggle_submission__f=0.8016696180116367_v=0.85
25564803804994.csv
```

```
Delta between Training and Validation is too high. Exiting loop.
```



Best Results vs Kaggle Unseen data



We can see from comparing our validation value to the public score on kaggle, that the training data has overfitted on the data quite a bit. 0.85255 validation accuracy on our test set, but only 0.7784 on the most recent score signifies quite a bit of overfitting, which was to be expected with the target encoding method.

Conclusion

I have found that the best results in the genetic programming model were achieved with a crossover probability of 0.4 and a mutation probability of 0.1. This combination consistently led to above average outcomes. The lower crossover probability of 0.4 seems to strike the right balance in mixing genetic information without causing too much disruption. At the same time, a somewhat high mutation rate of 0.1 helps introduce enough new traits to keep the solution evolving and improving. This specific setting seems to be the optimal mix for the model.

I have also found that Setup 2 (Encoder: target, Train:Test Split: 80:20) resulted in a higher number of better quality solutions compared to the other setups. Unfortunately, my testing was cut short, so going forward I will avoid making last minute changes that will impact evaluation.

Future explorations could include identifying and focusing on the most impactful features that could enhance model performance. Also, another option would be looking into methods to try and reduce overfitting, such as regularisation. I will also be looking into more advanced genetic techniques to see if further improvements can be made to the model.