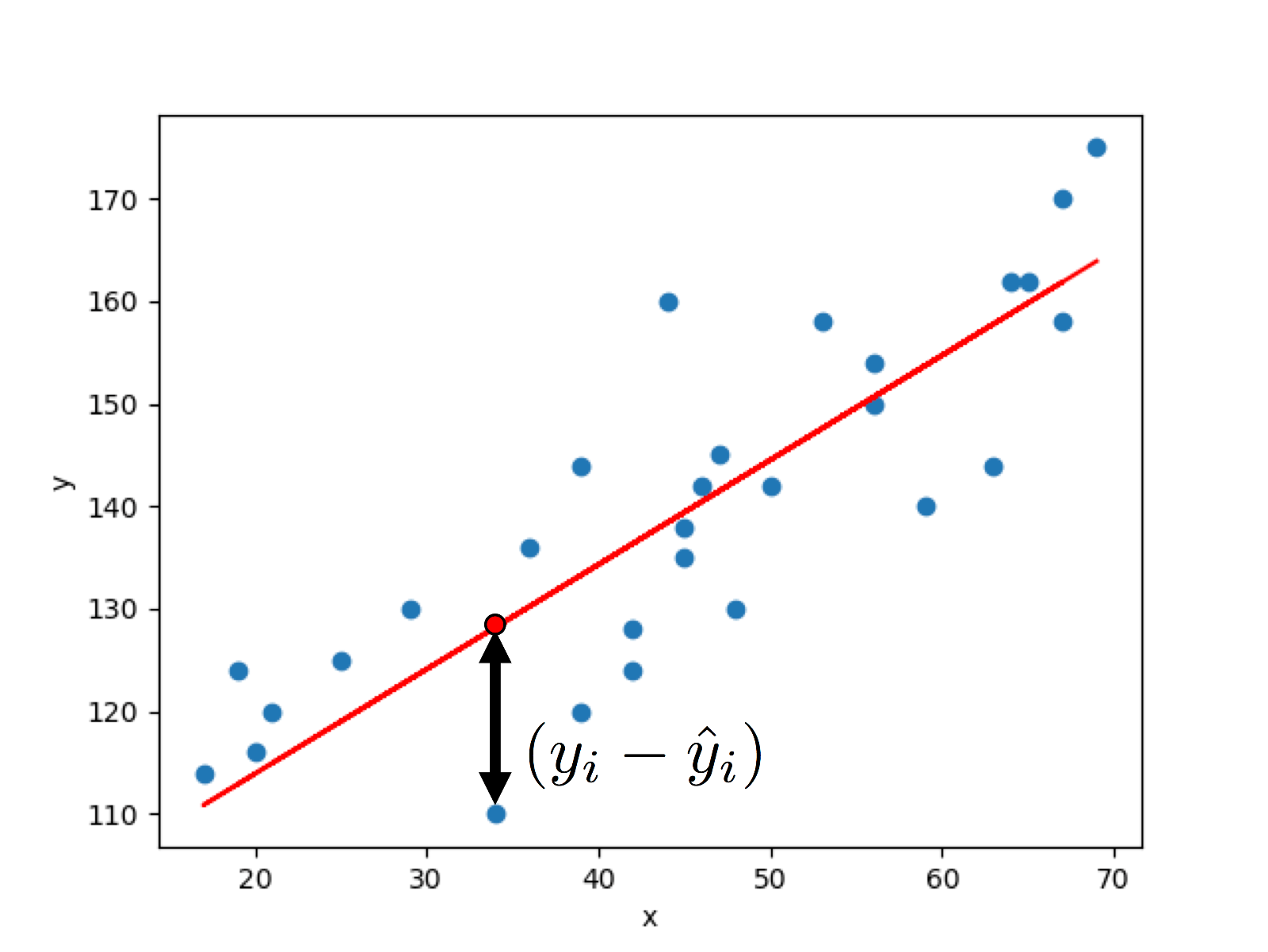
Computation - Final Project

Genetic algorithm

Jaime Resano Aísa

Spring 2023

Index

[1. Introduction 3](#_Toc136167881)

[1.1. Problem description 3](#_Toc136167882)

[1.2. Genetical algorithms 3](#_Toc136167883)

[1.3. Goals 3](#_Toc136167884)

[2. Genetic algorithm 4](#_Toc136167885)

[2.1. Implementation 4](#_Toc136167886)

[2.2. Functions 4](#_Toc136167887)

[3. GUI program 5](#_Toc136167888)

[3.1. How to use 5](#_Toc136167889)

[4. Conclusions 7](#_Toc136167890)

# Introduction

## Problem description

There is a training dataset available in CSV format, where each row represents an observation and the columns correspond to numerical input values, while the last column represents the outcomes. The input data consists of real numbers that have been normalized to facilitate their use.

## Genetical algorithms

In this project, a genetic algorithm will be implemented, which consists of the following main steps: creation of the initial population, calculation of individuals' fitness, creation of offspring from the initial parents, mutation of the offspring, and random selection of individuals for the next generation.

Firstly, an initial population is generated, consisting of a set of random individuals. Each individual represents a possible solution to the problem being addressed.

Next, the fitness of each individual in the initial population is evaluated. Fitness is a measure of the quality of each solution in terms of how well it fits the established optimization criteria. This calculation is performed using an evaluation function that assigns a numerical value to each individual based on their performance.

Afterwards, initial parents are selected for children creation. Parents are selected in pairs based on their fitness, meaning that individuals with better fitness are more likely to be chosen for reproduction. Then a crossover function generates a pair of individuals from them.

Then, mutation is applied to the generated children. Mutation is a process that introduces random changes in individuals in order to explore new regions of the search space and prevent premature convergence towards a suboptimal solution. Mutation alters the characteristics of individuals, allowing for greater genetic diversity in the population.

Finally, individuals for the next generation are randomly selected. This selection is done using a random sampling process, where each individual has a probability of being selected proportional to its fitness. This process ensures that fitter individuals have a higher probability of being chosen for the next generation, but also allows the inclusion of less fit individuals to maintain genetic diversity.

This process is repeated certain number of times or until a desired result is achieved.

## Goals

The purpose of this project is to apply a genetic algorithm to the training data to analyze the relationship between the input variables and the desired outcome.

* Implement a genetic algorithm that follows the steps of creating the initial population, calculating fitness, creating offspring from initial parents, mutating the offspring, and selecting individuals for the next generation.
* Develop an application with a graphical user interface that allows for the automatic execution of experiments. The graphical interface should be intuitive, configurable, and user-friendly, so that users can adjust algorithm parameters and define the problem to be solved.
* Design the application in a way that it can display and store an experiment log. The log should include relevant information about the conducted experiments.

# Genetic algorithm

## Implementation

All the code has been written in Python. The Numpy library has been used for all the operations of the genetic algorithm. The genetic algorithm has been implemented as a function called “run\_experiment” that receives all the input parameters required for the algorithm and returns a “result” object that contains the results of the experiment.

The input datasets are stored as files in CSV (comma separated values) format. They are located in the “dataset” folder.

This is a list of all the variables that determine a result:

* Number pseudo random generator seed: an integer
* Input dataset: a csv file
* Individual count: a positive integer
* Iterations: a positive integer
* Crossover probability: a real from 0 to 1
* Mutation probability: a real from 0 to 1
* Progenitor selection method: a function
* Crossover method: a function
* Mutation method: a function

## Functions

The genetic functions described in this section are stored in the “genetic\_functions.py” file.

The individuals have been modeled as an array of real numbers representing the weights of the linear regression model. The initial population is initialized as a matrix array with random values.

The fitness function takes all the input data and the population array and returns an array of real numbers which represent the fitness of each individual. The implementation chosen is the square error since it is simple and well-known.

The progenitor selection functions are stored in the “ProgenitorSelectionFunction” class. They are a “NamedFunction“ instance which is basically a custom class that acts like a function that has a name. There are three progenitor selection functions implemented:

* Roulete: Progenitors are chosen with a weighted random selection based on its fitness score.
* Tournament with replacement: The best progenitors of a subgroup are chosen.
* Tournament without replacement: Same as the previous function, but individuals that have already been chosen cannot be chosen again.

The mutation functions are implemented in the same way. There are two mutation functions implemented:

* Uniform: The mutated values are replaced by random values.
* Non-uniform: The individuals are taken in pairs and mutated by mixing their values. The mixing function is the arithmetic average but could be improved by taking into account the individual fitness.

The crossover methods are the following ones:

* Single point: The pair of individuals are mixed by “cutting” them in a single point.

A picture containing line, diagram, screenshot, rectangle

Description automatically generated

* A picture containing line, diagram, rectangle, colorfulness

  Description automatically generatedTwo points: There are two “cutting” points.

# GUI program

## How to use

The source code of the program can be found in the [GitHub repository](https://github.com/Jaime02/Genetic-Algorithm). In order to run it, the library PySide6 and other dependencies must have been installed and be available in the current Python interpreter. Installing all dependencies can easily be carried out by executing the following command:

pip install -r requirements.txt

Note that the command needs to be run in a terminal that has opened the project root folder, there it is the “requirements.txt” file that lists all dependencies.

A screen shot of a computer

Description automatically generated with medium confidenceThis is a screenshot of the program home page:

There is a central widget which is a table that has all the results of the experiments. It is sortable by clicking on the header of the column that needs to be sorted. There are two dock widgets that are placed on the window sides and can be moved and resized by the user. There is a list of checkboxes that toggle the visibility of the table columns. This has been implemented due to the number of columns required.

On the other dock, there are the experiment running options. The user can customize all of the experiment parameters. There has been implemented two modes of running experiments: the single experiment that runs the algorithm certain amount of times and the “combination mode”.

A screenshot of a computer

Description automatically generated with medium confidenceThe first mode runs the algorithm as many times as the “experiment count” input says. There are some parameters that are customizable and vary on each experiment. These parameters have an increase field that is applied on each experiment. The increase can also be negative. This is useful for testing how does the results change depending on different parameters.

The “combination mode” runs the experiments with the initial configuration but the functions are changed on each round. All of the possible combinations between the functions are tested. Since there are 3 progenitor selection functions, 2 mutation functions and 2 crossover functions, there are 3 \* 2 \* 2 = 12 possibilities. The experiments are also run as many times as the “experiment count” specifies. This feature shows how the results change depending on the functions chosen.

There is a plot for each result that shows the average fitness of each generation and the best fitness of it.

# Conclusions

A picture containing text, screenshot, plot, software

Description automatically generatedThe program plots the average and best fitness of each generation. An example plot is shown in the following picture.

The algorithm by design does not guarantee to find the best fitness possible. There are some improvements in the last generations. In the first iterations of the algorithm, the fitness of the population is improved substantially. In most cases, about 50 iterations is enough to get excellent fitness.

It is extremely hard to extract blunt conclusions about which parameters are better or worse than others since there are a lot of variables that affect the result of an experiment.

For example, let’s try to find out which is the best progenitor selection method. This are the experiment settings:

|  |  |
| --- | --- |
| A screenshot of a computer  Description automatically generated with medium confidence |  |
|  |  |

A screenshot of a computer

Description automatically generated with medium confidenceIn this case the roulette is worse than the tournament functions. The tournament functions are less stable but achieve a better result way faster. The fitness average stays between 0.05 and 0.1 quickly, whereas in the roulette case It requires more iterations. The tournament functions achieve 0.014 best fitness, but the roulette experiment reaches 0.0145.

Let’s try to do the same experiment with the crossover functions.

|  |  |
| --- | --- |
| A screen shot of a graph  Description automatically generated with medium confidence | A screen shot of a graph  Description automatically generated with medium confidence |

In this case, there are no noticeable differences between functions. However, this is just a single comparison that uses certain parameters. Is perfectly feasible that a different set up can show a difference between functions.

A screen shot of a computer

Description automatically generated with low confidenceThe mutation probability is an important factor that determines how important is the mutation function. This are the results of the experiment:

The results are remarkable since the optimal mutation value seems to be at 0.2 and increasing it causes worse results.

Finally let’s try all possible combinations of the functions. The dataset chosen is the Laser. The results are sorted by fitness in ascending order, so the best results are on top.

A screenshot of a computer

Description automatically generated with medium confidence

As stated previously, it is very hard to draw blunt conclusions due to the number of variables that affect the experiment. In this case the optimal result is at about 0.0135, but it can be improved by using more iterations and individual count. However, this makes the execution a lot slower.

The project has been completed successfully since all the goals have been completed. There are still a lot of features that could be added.

For example, the performance of the program could be improved by using a “Just in time” compiler like [Numba](https://numba.pydata.org/) which could speed up a lot the NumPy code.