IT3708: Project 1

Genetic Algorithms for Feature Selection using Python

Lab Goals:

- Implement a genetic algorithm to select the most impactful features in a dataset
- Implement crowding in the genetic algorithm to increase diversity within the population
- Show the results of the feature selection, and compare results when using crowding, no crowding and not doing any feature selection
- Use a simpler synthetic problem as a stepping stone towards the feature selection problem

Groups Allowed? Groups are allowed, max 2 persons. Every student must attend the demo day. The members of a group should sign up for the same time slot on demo day.

Deadline: February 09th, 2024

Assignment Background and Motivation

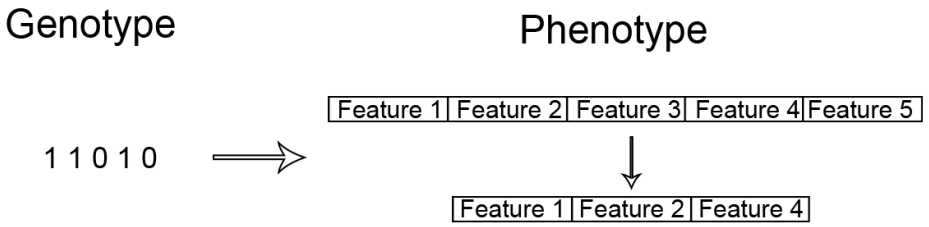
Feature selection is a machine learning process where redundant and irrelevant features are removed from a dataset [5]. Evolutionary algorithms including genetic algorithms have been successfully used for the purpose of feature selection [6]. The selected features will often provide approximately as good, or better, results, and having data with fewer dimensions has some advantages when training a predictive model:

- Faster training time
- Reducing overfitting
- Requiring less data entries

In this project you are tasked with creating a genetic algorithm for finding the best features for a given dataset. The features will be passed to a simple machine learning algorithm which returns the root-mean-square error. This root-mean-square error can in turn be used as a fitness score for the implementation of the genetic algorithm. The objective is to minimize the root-mean-square error. We will provide further details on this problem below, after discussing genetic algorithms.

Genetic Algorithms (GAs)

To solve this problem you will implement a genetic algorithm (GA) as presented in lectures and the literature. The so-called Simple Genetic Algorithm (SGA) is an excellent starting point [1] [2] [3]. It is recommended to represent the individuals as bitstrings. In the feature selection problem, each bit decides whether a feature should be included or not.

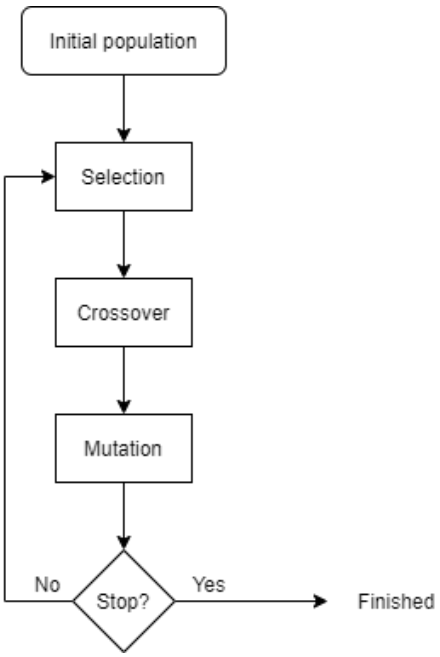


Note that GA parameter values (population size, generation number, crossover rate, mutation rate, etc.) are important to optimize and are typically correlated. Your GA may successfully find optimal or near-optimal solutions if you use reasonable parameter values. However, there is no definite rule as to how to find such parameter values. Therefore, you should test different sets of parameter values to decide on appropriate values. You are advised to closely study the interdependence between the parameter values in your chosen implementation.

There are many variants of genetic algorithms. In this project we will focus on two types: the simple genetic algorithm (SGA) [1] [2] [3] and the crowding algorithm [2] [4]. The two approaches are quite similar, but with some small but important differences in the replacement step.

GA 1: The Simple Genetic Algorithm

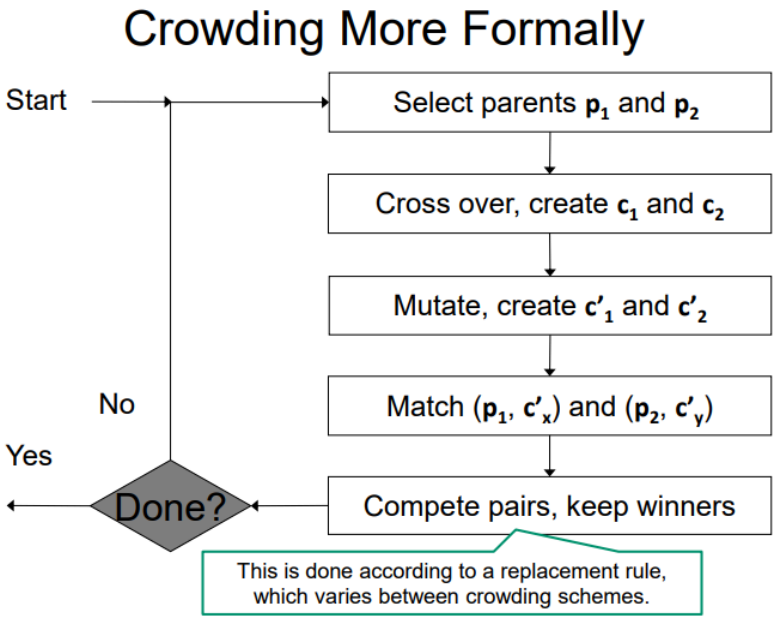
The simple genetic algorithm [1] [2] [3] consists of generating an initial population, selecting the best parents of the offspring, doing crossover and mutation, and replacing the population. This process is repeated until an end condition is reached.



GA 2: The Crowding Concept

A problem that may arise when using an evolutionary algorithm is having most or all individuals in the population converging to the same solution. This might be undesirable, because there might be better solutions that the algorithm is not finding due to premature convergence.

Crowding is a technique that can be applied to extend the SGA and achieve better performance. It combats the convergence to similar solutions by increasing diversity within the population [2] [4]. The survival step in the genetic algorithm is replaced with a local tournament, for example between a parent and its closest child. There are different variants of crowding, including De Jong's scheme, deterministic crowding, probabilistic crowding, and restricted tournament selection [2] [4]. In this project you are asked to implement and demonstrate one or two of these crowding methods (see below).

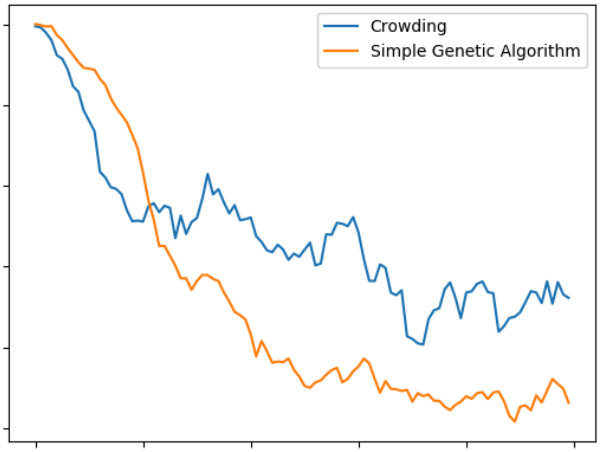


Population Diversity and Entropy

As explained above, it can be desirable to encourage a certain level of diversity or variation within the population of the genetic algorithm. To measure this variation, we can use the entropy of the population, given by the formula

$$H = -\sum_i p_i log_2 p_i$$

Here, *i* denotes the index in the bitstring, and p_i denotes the probability of the *i*th bit being a 1. The probability is estimated from the population.



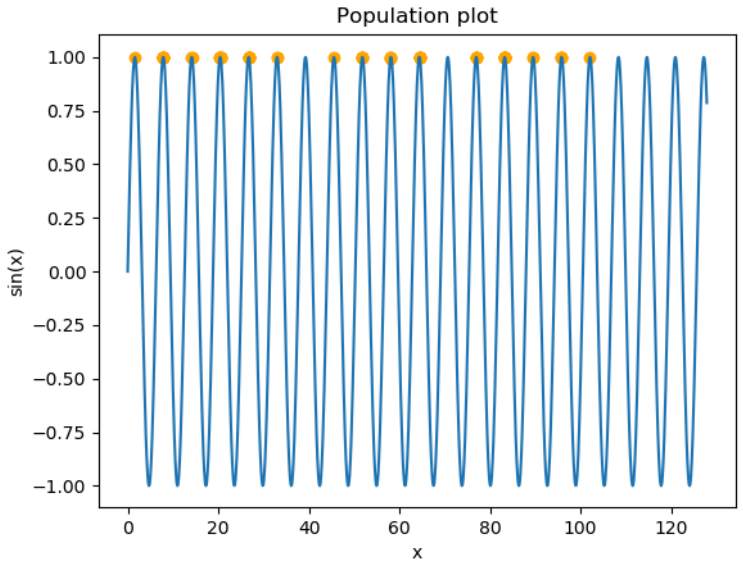
The figure shows a comparison of the entropies (on the *y*-axis) over the generations (on the *x*-axis) for implementations of a crowding GA and a simple GA. The figure suggests that the Crowding GA maintains diversity, as measured by entropy, better than the simple GA with increasing generations.

Fitness Functions and Tasks

In this project we will be implementing two different fitness functions, both of which operate on bit strings. Instead of directly attacking the feature selection problem, we will first study a synthetic or artificial problem. The practice of using simpler synthetic functions as stepping stones for more complex real-world problems is common in artificial intelligence theory and practice [2] [3] [4]. The GA stays essentially the same, perhaps some of the GA parameters need to be set differently, but the synthetic fitness function is faster to evaluate and easier to understand than natural fitness functions.

Fitness Function 1: Sine Function

The first fitness function we will be using is the sine function in the interval [0,128]. Here, the task will be to maximize the sine function on this interval. In order to pass your bitstring into the sine function, you will first have to convert it into a real value and then scale it to fit into the interval. For example will a bitstring of size 15 have a maximum value of 2^{15}, which gives us a scaling factor of 2^{-8}. The plot below demonstrates how a crowding GA works well in optimizing this fitness function while maintaining diversity.



Plot of the fitness function sin(x) and the individuals(orange) in the population

Indirect Constraint Handling with Penalty Functions

Constraints can be handled indirectly by adding a penalty term to the fitness function. The original fitness function f(x) is then transformed into f'(x) = f(x) + P(d(x,F)), where d(x,F) is a distance metric of the infeasible point to the feasible region F. The penalty function P is zero for feasible solutions and increases proportionally with the distance to the feasible region [1].

Fitness Function 2: Feature Selection

The second task will be related to the feature selection task described above. Two key components of this task are a *dataset* as well as a *machine learning algorithm*. Both of these components are provided as part of the project assignment and are also presented below.

Machine learning algorithm. The provided file for this project, **LinReg.py**, contains a class that will run a simple machine learning algorithm (linear regression) for a dataset, and return the root-mean-squared error. This error can be used as a fitness function for a genetic algorithm. It is important to keep in mind that your genetic algorithm should attempt to minimize the root mean squared error, and not maximize it. You don't need to understand the details of linear regression in order to use the root mean squared error as a fitness function.

In order to use the **LinReg.py** file, you will need Numpy and scikit-learn. These are some useful functions provided in the LinReg.py file as well:

- ***get_fitness(x,y)***: Gives the root-mean-square error on the data ***x*** with targets ***y***
- ***get_columns(x, bitstring)****:* Selects features from data ***x*** based on the bits in ***bitstring***

You can learn more about these methods by looking at the documentation in the code.

Dataset. The dataset to be used for machine learning and feature selection consists of 1994 rows each having 102 columns. The first 101 columns represent the data, while the last column represents the value of the row. When calling get_fitness in the LinReg class supplied, the data and values should be passed as parameters like this: get_fitness(data, values). The LinReg class also contains a helper function to filter out the columns by providing it with the data and a bitstring. Usage: get_columns(data, bitstring).

Programming (12 points maximum)

When implementing the different functions it is a good idea to keep the GA parameters in one place. You might be asked to change these values during the demonstration of the assignment. Here are the tasks you need to implement and be ready to demonstrate and answer questions about during the demo day:

For the sine synthetic problem

a) Implement a function to generate an initial population for your genetic algorithm. **(0.5p)**

b) Implement a parent selection function for your genetic algorithm. This function should find the fittest individuals in the population, and select parents based on this fitness. **(0.5p)**

c) Implement a function that creates two offspring from two parents through crossover. The offspring should also have a chance of getting a random mutation. **(0.5p)**

d) Implement survivor selection. **(0.5p)**
   - For a *1-person team*: Implement one survivor selection function that selects the survivors of a population based on their fitness.
   - For a *2-person team*: Implement two such survivor selection functions.

e) Connect all the implemented functions to complete the genetic algorithm, and run the algorithm with the sine fitness function. Throughout the generations plot the individuals, values and fitness values with the sine wave. **(2p)**

f) This task is identical to *e)*, however, we now add the constraint that the solution must reside in the interval [5,10]. The constraint should not be handled by scaling the real value of the bitstring into the new interval. Throughout the generations plot the individuals, values and fitness values with the sine wave. **(2p)**

For the feature selection problem

g) Run the genetic algorithm on the provided dataset. Show the results, and compare them to the results of not using any feature selection (given by running the linear regression with all features selected). The points given here depend on the achieved quality of the result and team size. **(3p)**
   - For a *1-person team* RMSE less than 0.125.
   - For a *2-person team* RMSE less than 0.124

h) Implement a new survivor selection function. This function should be using a crowding technique as described in the section about crowding. Do exercise f) and g) again with the new selection function, and compare the results to using the simple genetic algorithm. Also show and compare how the entropies of the different approaches (SGA and crowding) change through the generations through a plot. **(3p)**
   - For a *1-person team*: implement and demonstrate one crowding approach.
   - For a *2-person team*: implement and demonstrate two crowding approaches.

Theory Questions (3 points points maximum)

The following are examples of questions that you may be asked during demo-day.

a) What is niching in the context of evolutionary algorithms, and is it helpful in this project?
b) What's the difference between the genotype and phenotype in this project?

Note, you will probably not be asked exactly these questions, and the questions may be more tailored to your particular implementation and demonstration. Further, the number and/or difficulty of the questions will be greater for a 2-person team than for a 1-person team.

Delivery Method and Deadline

You should deliver a zip file with your code on BlackBoard. The submission system will be closed on February 09th. The demo day for Project 1 is also February 09th. A signup schedule will be announced one week before. Please follow Blackboard for details.

Every student must submit their (jointly) developed code. You must attend the demo individually on the scheduled demo date. No early or late submission or demo will be entertained except in an emergency.

References

[1] A. E. Eiben and J. E. Smith. “Introduction to Evolutionary Computing,” 2nd Edition, Springer 2015, pages 99 - 100, Table 6.1 (SGA) & pages 91 - 95 (population diversity, crowding) & pages 203 - 211 (constraint handling).
[2] D. Simon. “Evolutionary Optimization Algorithms,” Wiley 2013, pages 44 - 55 (SGA) & pages 192 - 198 (population diversity, crowding).
[3] D. E. Goldberg “Genetic Algorithms in Search, Optimization, and Machine Learning,” Addison-Wesley, 1989, pages 59-75.
[4] O. J. Mengshoel and D. E. Goldberg. 2008. “The Crowding Approach to Niching in Genetic Algorithms.” Evol. Comput. 16, 3 (Fall 2008), pages 315–354. DOI:https://doi.org/10.1162/evco.2008.16.3.315
[5] I. Guyon and A. Elisseeff. 2003. “An Introduction to Variable and Feature Selection.” Journal of Machine Learning Research 3, (3/1/2003), pages 1157–1182.
[6] B. Xue, M. Zhang, W. N. Browne and X. Yao, "A Survey on Evolutionary Computation Approaches to Feature Selection." IEEE Transactions on Evolutionary Computation, vol. 20, no. 4, pages 606-626, Aug. 2016, doi: 10.1109/TEVC.2015.2504420.