Training of genetic algorithms

Fernando Murillo Bravo
Sevilla, España
fermurbra@alum.us.es

I. Introduction

I have developed a genetic algorithm to solve a regression problem using a dataset of cars. This dataset url contains various attributes of cars, such as specifications, features, and ultimately the price of each car. To handle and preprocess the dataset effectively, I utilized Apache Spark, which allowed me to manage and manipulate the data efficiently for the purposes of the genetic algorithm.

The project is implemented in Python and is organized into four main components: the Chromosome class, which represents individual solutions; the CSV Reader, which handles the reading and preprocessing of the dataset; the Genetic Algorithm, which performs the optimization process; and the Main, where the overall execution and integration of these components take place.

The genetic algorithm is applied to develop a regression model, with the objective of minimizing the error between the predicted and actual car prices. The performance of the model is evaluated using the Root Mean Square Error (RMSE), a widely used metric for assessing the accuracy of regression models. This approach ensures a robust and scalable solution for the predictive modeling task.

II. Metodology

Now, we will explain in full detail how the algorithm works on a technical level, describing the purpose of each function and discussing potential technical considerations, design decisions, or challenges encountered.

To begin, in this work, we are using genetic algorithms to obtain an optimal solution to a regression problem. Our solution models relationships between certain numerical variables we know to provide a numerical prediction: we use a model that relates the input attributes $x_1, x_2, \ldots, x_{d-1}, x_d$ to model an output $\hat{y}$. We follow the following mathematical expression:

$$\hat{y} = c_1 \cdot x_1^{e_1} + \cdots + c_d \cdot x_d^{e_d} + C \tag{1}$$

Where $\hat{y}$ is the number predicted by our solution for those attributes, c_i is the coefficient that multiplies the attribute x_i, e_i is the exponent to which the base x_i is raised, and C is the final coefficient added to all the above.

In this way, we can see that for each solution it is necessary to store a coefficient and an exponent for each attribute, plus a final coefficient C. Therefore, we model the chromosome as a list of d+1 floating-point numbers, where d is the number of known attributes. If we have 17 attributes, we will have 2*17+1 = 35 genes, which will be encoded as follows.

In our genetic algorithm, each chromosome is represented as a list of floating-point numbers. These floating-point numbers are organized in such a way as to define the coefficients and exponents of the input attributes in our regression model. The structure of this list is described below:

Chromosome structure

The list of float numbers for a chromosome with d attributes of entry $x_1, x_2, \ldots, x_d$ can be represented as:

$$[c_1, e_1, c_2, e_2, \ldots, c_d, e_d, C]$$

Each number of this list is a float, and, given an index i to reference the attribute x_i, its coefficient will be $chromosome_{i*2}$ and its exponent will be $chromosome_{i*2+1}$. Therefore, the prediction of a chromosome for a set of attributes $x_1, \ldots, x_d$ is:

$$\sum_{i=0}^{d-1} chromosome_{i*2} * x_i^{chromosome_{i*2+1}} + C$$

But this way of calculating the prediction of an individual can present problems when the attribute x_d we receive is negative and the exponent $chromosome_{i*2+1}$ is fractional, as the solution in this case, applying the formula explained, results in a complex number. Since we are looking for a numerical solution in the real plane, this is not acceptable. For this reason, we have explored various solutions to this problem, where the complex number obtained is approximated to a real number without losing too much information.

What has been explained in this section is the structure of the chromosomes and the computeY method. This function, given a chromosome and a set of attributes, provides the prediction $\hat{y}$ of that specific individual for the given set of attributes.

Generate initial population

To initiate our algorithm and begin evolving generations, we need to start with an initial population, which we will generate randomly to fulfill the purpose mentioned previously. Subsequently, new generations of chromosomes will be obtained through crossovers, random mutations, and the random generation of the initial population.

Since our chromosomes are lists of $d * 2 + 1$ numbers, where d is the size of each attribute set, we will randomly generate $d * 2 + 1$ numbers using Python's 'random.uniform' function. This process will be repeated as many times as the number of individuals we want in our population, generating a chromosome each time and adding it to our population list (a list of lists of floats).

In our project, the generatePopulation function generates the initial population of chromosomes. It does not take any explicit parameters, as it relies on the predefined number of individuals (self.nInd) and the fixed chromosome size (35, corresponding to the 35 attributes in the dataset, $d * 2 + 1$). The function creates as many individuals as specified by the variable n and generates each individual using the createRandom function. This function returns a chromosome consisting of 35 floating-point numbers, all of which are randomly generated within the range of -1 and 1.

Calculation of the fitness of each individual

Once we are able to make a prediction of $\hat{y}$ for a given individual, we must use this calculated prediction to determine the fitness of that individual, more commonly known as the fitness function.

This fitness or fitness value is a number that evaluates how well the individual performs across all attribute sets. To do this, for each set of attributes, we make a prediction $\hat{y}$ and compare it with the expected y. There are various techniques to determine the fitness of a solution given these two numbers, but we have used RMSE, such that the fitness of a chromosome is given by the following formula:

$$\text{RMSE} = \sqrt{\frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} (\hat{y} - y)^2}$$

Where $|\mathcal{D}|$is the number of sets or number of predictions made..

Thus, the fitnessFunction method, which receives a chromosome and all data sets, first finds the set of predicted solutions, and then compares them with the correct solutions, using the RMSE explained.

It is important to note that, since the RMSE gives the error, and fitness is calculated directly from this measure, for an individual to have a numerically larger fitness value means that it has a larger error, and is worse.

This is relevant to keep in mind when understanding the selection methods explained below: when we say that an individual has a good fitness value, or that its fitness is better, we are saying that the numerical value returned by its fitness function is small or that it is smaller, and vice versa.

Population evolution: elite and parental selection methods

Once we know how to calculate, for each individual, their fitness value, we are interested in moving the population towards better fitness values. This, as explained when we discussed the nature of the algorithm, is done by ensuring that an elite population (those with the best fitness values) moves on to the next, and, among the rest, selecting parents (by tournament) by promoting those with the best fitness, and mixing the genetic material as explained in the next point.

When selecting an elite we start from a hyperparameter $elitismRate$, which indicates the relative number of individuals in the elite. If, for example, this hyperparameter is 0.1 and the population is 100, there will be 10 members in the elite.

These members will pass directly unchanged to the next generation, ensuring that the best individual of a generation never becomes worse than the previous one: if, through the randomness of crossovers and mutation, the bulk of the individuals become worse, the elite ensures that the fitness of the best individual of the generation remains as good.

Once we have passed the elite to the next generation, we will search among the remaining individuals for parents, and we will use the tournament selection method, following a k hyperparameter.

This method, parentSelection, takes as parameters the population evaluated and sorted by its fitness value (after having removed the elite set), and the aforementioned k. From the past population, k individuals are randomly selected, and the one with the best fitness is added to a new set of parents.

This is repeated until there are as many parents as there are chromosomes in the population we have passed on, and, in the set of parents, there may be some repeated chromosomes (in fact, that is what we intend, because otherwise the bad fitness chromosomes will have as many offspring as the good fitness ones and we would not be promoting the offspring of the good ones).

Finally, from the set of parents we will randomly select pairs of 2, cross them, get two offspring, perhaps mutate them, and add them to the new population. Finally, we remove these two chromosomes from the set and repeat until there are no chromosomes left in the parent set, and therefore the new population has the same size as the previous population.

Method of crossover

Another fundamental method of genetic algorithms is the crossover of good solutions.

To achieve this, we combine solutions with good fitness with the goal of producing offspring that inherit favorable genes from both parents, thereby further improving their fitness.

The crossover function takes as parameters a list of two chromosomes (the parents) and a crossover probability p. If the established probability p occurs, both parents are crossed using the single-point crossover operator: a crossover point is chosen randomly, and this point p represents the index within each child. Child 1 inherits the genes of parent 1 up to index p, and from parent 2 starting at index p, while child 2 inherits the opposite pattern.

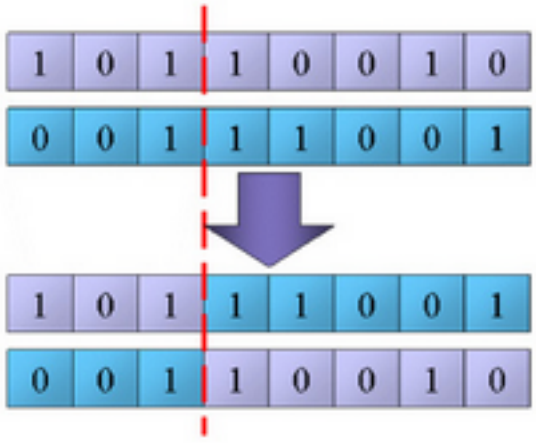

Fig. 1. cross point

Method of mutation

Finally, another fundamental method is mutation. This technique aims to explore new solutions in the

solution space close to the individuals we already have, expanding the regions we explore and reaching areas that we might not have reached otherwise.

These mutations involve making small changes to some individuals.

In terms of implementation, the 'mutation' function takes as parameters a chromosome, the mutation probability $mutationProb$, and the mutation intensity d (the latter two are hyperparameters). For each gene in the chromosome, if the mutation probability occurs, a random number in the range $(-d, d)$ is generated and added to the value of that gene.

If the mutation probability does not occur, the gene remains unchanged.

III. Results

Having explained all the code, we continue with the results we have obtained after the tests:

We have worked with the trainUpdated file to get the best chromosome, the one with the lowest RMSE when predicting the training data. This chromosome with the best fitness (lowest error) was used to test how accurate it is with the testUpdated file.

We have used the following parameters: value of elements of chromosome [-1,1], number of individuals = 30, number of iterations = 40, elitism rate = 0,1 , parent selection rate per tournament = 3, crossover probability = 0,7, mutation probability = 0,1 and mutation = 0,3 by default.

Although we have previously been testing these parameters to improve our results, we have come to the following conclusions:By increasing the range of possible values in the chromosome randomly between -10 and 10, the RMSE goes up.
By increasing the parent selection rate per tournament to 4 we also see the RMSE increase, because the parents start to repeat. Decreasing the crossover probability clearly worsens the RMSE result, because there is less variety of solutions due to fewer offspring.

We have obtained the following results:

Generation 0 RMSE: (2.011211462816065)
Generation 10 RMSE: (0.6061075188820919)
Generation 20 RMSE: (0.5125245077200907)
Generation 30 RMSE: (0.4062776969880182)
Generation 39 RMSE: (0.3261757179750217)

test: RMSE 0.32133231036897586

We may wonder why from the beginning the RMSE is so low. This is because all the values of the dataset have been modified to make it easy to use. With Spark we have managed to obtain a normalised and encoded dataset, keeping all its attributes between [0,1].

IV. Conclusion

This work focused on the development and evaluation of a genetic optimisation algorithm to solve regression problems. Several selection, crossover and mutation techniques were implemented to improve the efficiency of the algorithm. Experiments showed that the algorithm improves significantly as the iterations progress.

Based on testing the training and test file with different values of the hyperparameters n, numGen, elitismRate, p, k, mutationProb and mutationDelta, the parameters set in the project have been left as they are.

The results obtained indicate that the use of mutations that modify the individual to the right extent increases genetic diversity in the population, which in turn leads to a better exploration of the solution space. Furthermore, it was observed that the implementation of elitism ensures the retention of the best solutions over generations, avoiding degradation of the population quality, and that using a random crossover point increases the genetic variability of the solutions. These findings confirm that combining advanced genetic optimisation strategies can result in more optimal and efficient solutions.