Hyperparameter tuning in XGBoost using genetic algorithm

Mohit Jain
Sep 17, 2018 · 9 min read

. . .

Introduction

Genetic algorithm, as defined in wikipedia, takes its inspiration from the process of natural selection, proposed by Charles Darwin. In a more general term, we can understand the natural process and how it is related it to the genetic algorithm, using the following description:

We start with initial population, which will have certain traits, as shown in Figure 1. This initial population will be tested in a certain environment to observe how well the individuals (parents) in this population perform, based on a predefined fitness criteria. The fitness in case of machine learning can be any performance metric- accuracy, precision, recall, F1-score, auc among others. Based on the fitness value we select top performing parents (“survival of the fittest”), as the survived population (Figure 2).

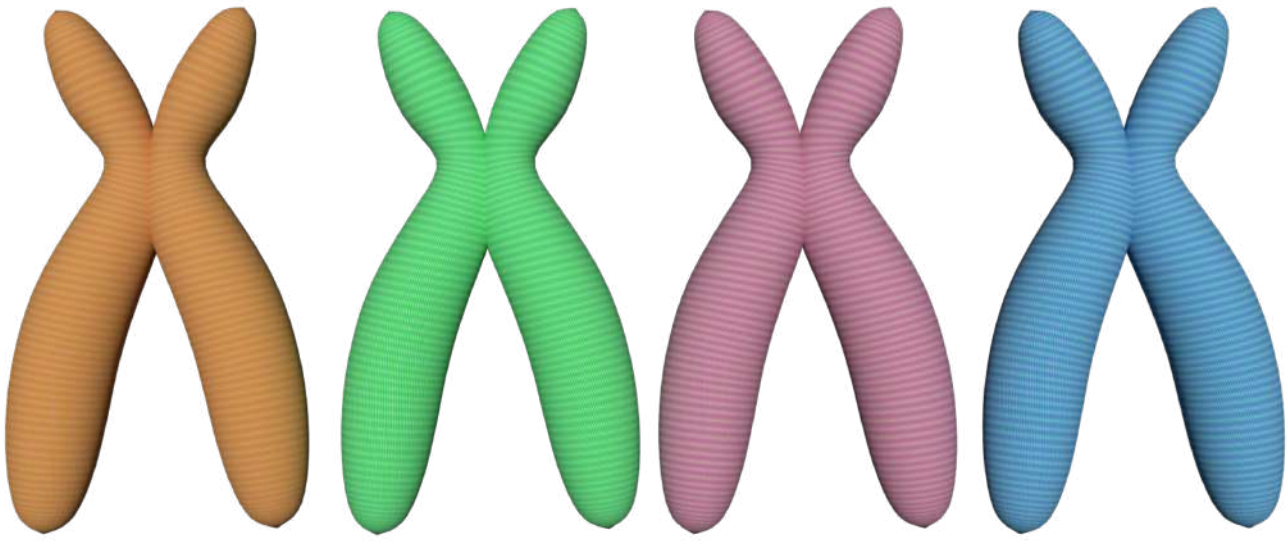


Figure 1: Initial Population

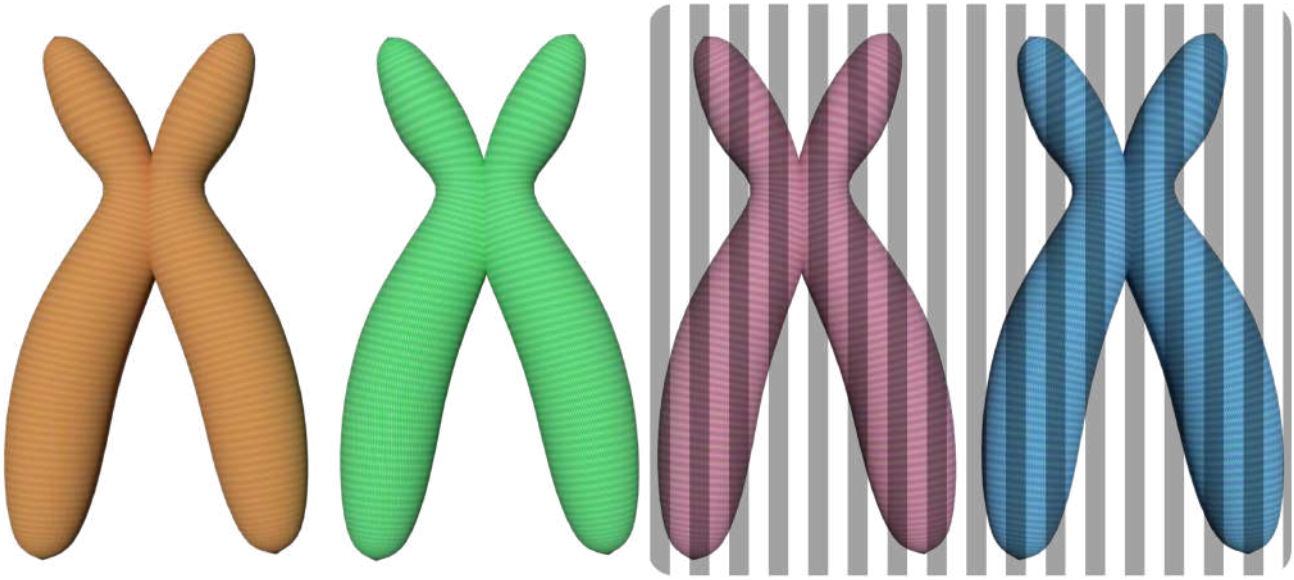


Figure 2: Survivied Population

Now the parents in the survived population will mate to produce offspring, using a combination of two steps: crossover/recombination and mutation. In the case of crossover, the genes (parameters) from the mating parents will be recombined, to produce offspring, with each children inheriting some genes (parameters) from each parent (Figure 3).

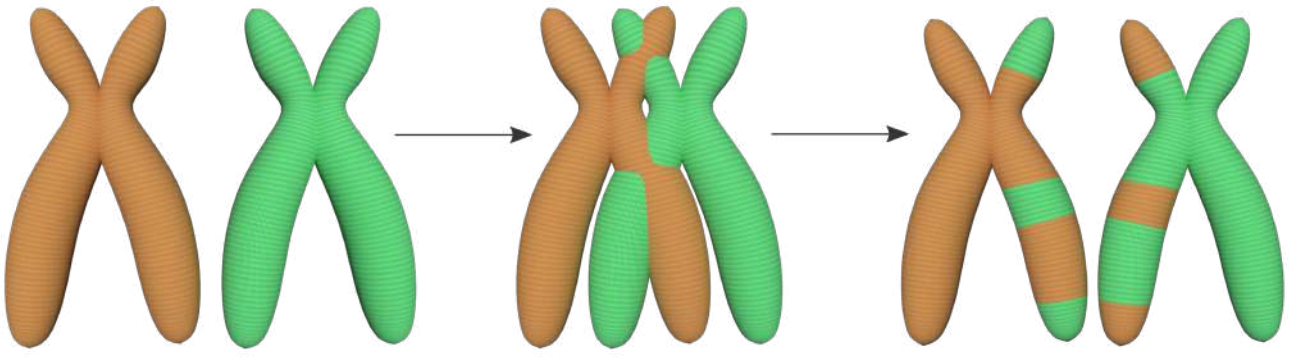


Figure 3: Crossover

Finally, in the case of mutation, some of the values of the genes (parameters) will be altered to maintain a genetic diversity (Figure 4). This allows the nature/genetic algorithm to typically arrive at a better solution.

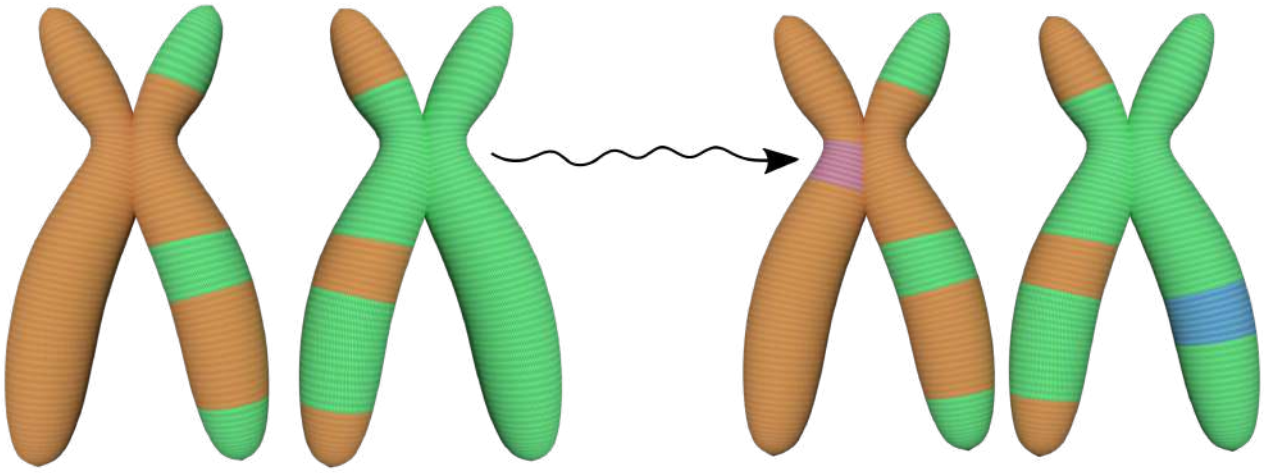


Figure 4: Mutation

Figure 5 shows the second generation of population, which will contain both survived parents as well as children. We keep the survived parents so as to retain best fitness parameters in case the offspring's fitness value turns out to be worse than the parents.

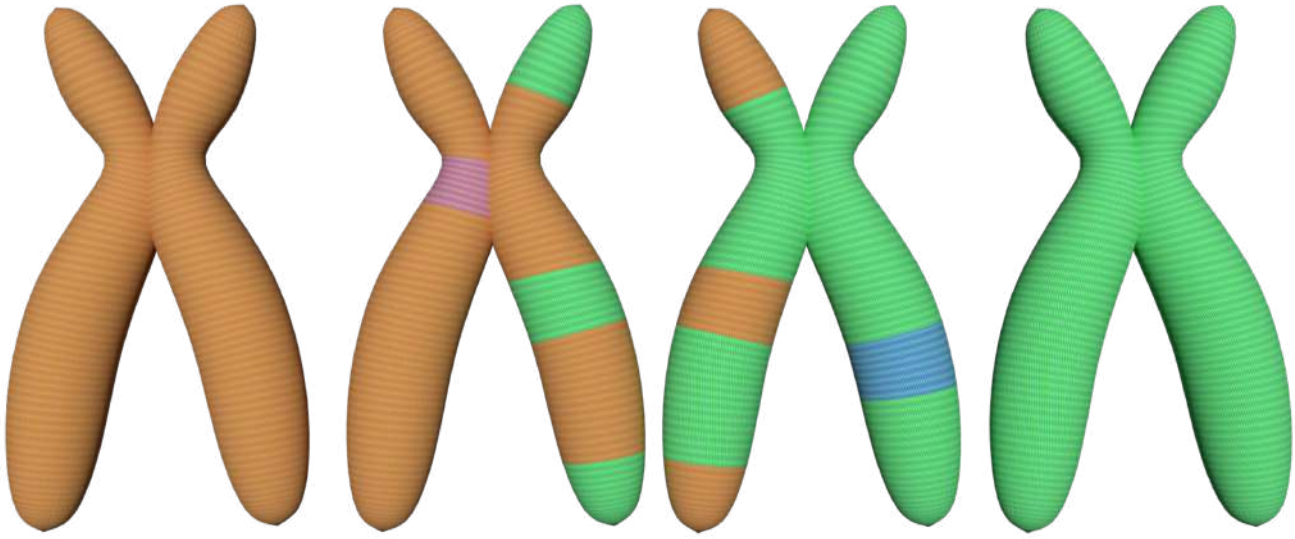


Figure 5: Second Generation

Genetic Algorithm Module for XGBoost:

We will create a genetic algorithm module customized for XGBoost. Below is the description of XGboost:

> ***XGBoost*** *is an optimized distributed gradient boosting library designed to be highly* ***efficient***, ***flexible*** *and* ***portable****. It implements machine learning algorithms under the*

The module will have functions to follow the four steps: (i) initialization, (ii) selection, (iii) crossover, and (iv) mutation, similar to the what is discussed above (A small section of this code is inspired from the post here).

Initialization:

The first step is initialization, where the parameters are randomly initialized to create the population. It is similar to the first generation of the population shown in Figure 1. The code below shows the initialization process, where we generate a vector containing the parameters. In the case of XGBoost we have selected 7 parameters to optimize: learning_rate, n_estimators, max_depth, min_child_weight, subsample, colsample_bytree, and gamma. A detailed description of these parameters can be found here.

```
def initilialize_poplulation(numberOfParents):
    learningRate = np.empty([numberOfParents, 1])
    nEstimators = np.empty([numberOfParents, 1], dtype = np.uint8)
    maxDepth = np.empty([numberOfParents, 1], dtype = np.uint8)
    minChildWeight = np.empty([numberOfParents, 1])
    gammaValue = np.empty([numberOfParents, 1])
    subSample = np.empty([numberOfParents, 1])
    colSampleByTree =  np.empty([numberOfParents, 1])

for i in range(numberOfParents):
        print(i)
        learningRate[i] = round(random.uniform(0.01, 1), 2)
        nEstimators[i] = random.randrange(10, 1500, step = 25)
        maxDepth[i] = int(random.randrange(1, 10, step= 1))
        minChildWeight[i] = round(random.uniform(0.01, 10.0), 2)
        gammaValue[i] = round(random.uniform(0.01, 10.0), 2)
        subSample[i] = round(random.uniform(0.01, 1.0), 2)
        colSampleByTree[i] = round(random.uniform(0.01, 1.0), 2)

    population = np.concatenate((learningRate, nEstimators,
maxDepth, minChildWeight, gammaValue, subSample, colSampleByTree),
axis= 1)
    return population
```

The limits on the parameters are either based on the limits described in the XGBoost documentation or based on an reasonable guess, if the upper limit is set to infinity. We start with creating an empty array for each parameter, followed by populating it with random values.

Parent selection (survival of the fittest)

In the second step we train our model using the initial population and calculate the fitness value. In this case we will calculate the F1-score.

```
def fitness_f1score(y_true, y_pred):
    fitness = round((f1_score(y_true, y_pred, average='weighted')),
4)
    return fitness

#train the data annd find fitness score
def train_population(population, dMatrixTrain, dMatrixtest, y_test):
    fScore = []
    for i in range(population.shape[0]):
        param = { 'objective':'binary:logistic',
              'learning_rate': population[i][0],
              'n_estimators': population[i][1],
              'max_depth': int(population[i][2]),
              'min_child_weight': population[i][3],
              'gamma': population[i][4],
              'subsample': population[i][5],
              'colsample_bytree': population[i][6],
              'seed': 24}
        num_round = 100
        xgbT = xgb.train(param, dMatrixTrain, num_round)
        preds = xgbT.predict(dMatrixtest)
        preds = preds>0.5
        fScore.append(fitness_f1score(y_test, preds))
    return fScore
```

We will define how many parents we would like to select and create an array with the selected parents, based on their fitness value.

```
#select parents for mating
def new_parents_selection(population, fitness, numParents):
    selectedParents = np.empty((numParents, population.shape[1]))
#create an array to store fittest parents

    #find the top best performing parents
    for parentId in range(numParents):
        bestFitnessId = np.where(fitness == np.max(fitness))
        bestFitnessId  = bestFitnessId[0][0]
        selectedParents[parentId, :] = population[bestFitnessId, :]
        fitness[bestFitnessId] = -1 #set this value to negative, in
case of F1-score, so this parent is not selected again
    return selectedParents
```

Crossover

There are various methods to define crossover in the case of genetic algorithms, such as single-point, two-point and k-point crossover, uniform crossover and crossover for ordered lists. We are going to use uniform crossover, where each parameter for the child will be independently selected from the parents, based on a certain distribution. In our case we will use “discrete uniform” distribution from numpy random function .

```
'''
Mate these parents to create children having parameters from these
parents (we are using uniform crossover method)
'''
def crossover_uniform(parents, childrenSize):

    crossoverPointIndex = np.arange(0, np.uint8(childrenSize[1]), 1,
dtype= np.uint8) #get all the index
    crossoverPointIndex1 = np.random.randint(0,
np.uint8(childrenSize[1]), np.uint8(childrenSize[1]/2)) # select
half  of the indexes randomly
    crossoverPointIndex2 = np.array(list(set(crossoverPointIndex) -
set(crossoverPointIndex1))) #select leftover indexes

    children = np.empty(childrenSize)

    '''
    Create child by choosing parameters from two parents selected
using new_parent_selection function. The parameter values
    will be picked from the indexes, which were randomly selected
above.
    '''
    for i in range(childrenSize[0]):

        #find parent 1 index
        parent1_index = i%parents.shape[0]
        #find parent 2 index
        parent2_index = (i+1)%parents.shape[0]
        #insert parameters based on random selected indexes in
parent 1
        children[i, crossoverPointIndex1] = parents[parent1_index,
crossoverPointIndex1]
        #insert parameters based on random selected indexes in
parent 1
        children[i, crossoverPointIndex2] = parents[parent2_index,
crossoverPointIndex2]
    return children
```

Mutation

The final step will be to introduce diversity in the children by randomly selecting one of the parameter and altering it's value by a random amount. We will introduce some limits also, in order to constrain the altered values within a certain range. Skipping these constrains might lead to error.

```
def mutation(crossover, numberOfParameters):
    #Define minimum and maximum values allowed for each parameter

minMaxValue = np.zeros((numberOfParameters, 2))

    minMaxValue[0:] = [0.01, 1.0] #min/max learning rate
    minMaxValue[1, :] = [10, 2000] #min/max n_estimator
    minMaxValue[2, :] = [1, 15] #min/max depth
    minMaxValue[3, :] = [0, 10.0] #min/max child_weight
    minMaxValue[4, :] = [0.01, 10.0] #min/max gamma
    minMaxValue[5, :] = [0.01, 1.0] #min/maxsubsample
    minMaxValue[6, :] = [0.01, 1.0] #min/maxcolsample_bytree

    # Mutation changes a single gene in each offspring randomly.
    mutationValue = 0
    parameterSelect = np.random.randint(0, 7, 1)
    print(parameterSelect)
    if parameterSelect == 0: #learning_rate
        mutationValue = round(np.random.uniform(-0.5, 0.5), 2)
    if parameterSelect == 1: #n_estimators
        mutationValue = np.random.randint(-200, 200, 1)
    if parameterSelect == 2: #max_depth
        mutationValue = np.random.randint(-5, 5, 1)
    if parameterSelect == 3: #min_child_weight
        mutationValue = round(np.random.uniform(5, 5), 2)
    if parameterSelect == 4: #gamma
        mutationValue = round(np.random.uniform(-2, 2), 2)
    if parameterSelect == 5: #subsample
        mutationValue = round(np.random.uniform(-0.5, 0.5), 2)
    if parameterSelect == 6: #colsample
        mutationValue = round(np.random.uniform(-0.5, 0.5), 2)

    #indtroduce mutation by changing one parameter, and set to max
or min if it goes out of range
    for idx in range(crossover.shape[0]):
        crossover[idx, parameterSelect] = crossover[idx,
parameterSelect] + mutationValue
        if(crossover[idx, parameterSelect] >
minMaxValue[parameterSelect, 1]):
            crossover[idx, parameterSelect] =
minMaxValue[parameterSelect, 1]
        if(crossover[idx, parameterSelect] <
minMaxValue[parameterSelect, 0]):
            crossover[idx, parameterSelect] =
minMaxValue[parameterSelect, 0]
    return crossover
```

Implementation

We will implement the module discussed above to train on a dataset. The dataset is from UCI Machine Learning Repository. It contains a set of 102 molecules, out of which 39 are identified by humans as having odor that can be used in perfumery and 69 not having the desired odor. The dataset contains 6,590 low-energy conformations of these molecules, containing 166 features. We are doing minimal prepossessing as the goal of this tutorial to understand genetic algorithm.

```
# Importing the libraries
import numpy as np
import pandas as pd
import geneticXGboost #this is the module we crated above
import xgboost as xgb

np.random.seed(723)

# Importing the dataset
dataset = pd.read_csv('clean2.data', header=None)

X = dataset.iloc[:, 2:168].values #discard first two coloums as
these are molecule's name and conformation's name

y = dataset.iloc[:, 168].values #extrtact last coloum as class (1 =>
desired odor, 0 => undesired odor)

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.20, random_state = 97)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

#XGboost Classifier

#model xgboost
#use xgboost API now
xgDMatrix = xgb.DMatrix(X_train, y_train) #create Dmatrix
xgbDMatrixTest = xgb.DMatrix(X_test, y_test)
```

We have 8 parents to start with and we select 4 fittest parents for mating. We will create 4 generations and monitor the fitness (F1-score). Half of the parents in the subsequent generation will be the fittest parents selected from the previous generation.

This will allow us to keep the best fitness score at least same as the previous generation, in case the children fitness score is worse.

```
numberOfParents = 8 #number of parents to start
numberOfParentsMating = 4 #number of parents that will mate
numberOfParameters = 7 #number of parameters that will be optimized
numberOfGenerations = 4 #number of genration that will be created

#define the population size

populationSize = (numberOfParents, numberOfParameters)

#initialize the population with randomly generated parameters
population =
geneticXGboost.initilialize_poplulation(numberOfParents)

#define an array to store the fitness  hitory
fitnessHistory = np.empty([numberOfGenerations+1, numberOfParents])

#define an array to store the value of each parameter for each
parent and generation
populationHistory =
np.empty([(numberOfGenerations+1)*numberOfParents,
numberOfParameters])

#insert the value of initial parameters in history
populationHistory[0:numberOfParents, :] = population

for generation in range(numberOfGenerations):
    print("This is number %s generation" % (generation))

    #train the dataset and obtain fitness
    fitnessValue =
geneticXGboost.train_population(population=population,
dMatrixTrain=xgDMatrix, dMatrixtest=xgbDMatrixTest, y_test=y_test)
    fitnessHistory[generation, :] = fitnessValue

    #best score in the current iteration
    print('Best F1 score in the this iteration =
{}'.format(np.max(fitnessHistory[generation, :])))

#survival of the fittest - take the top parents, based on the
fitness value and number of parents needed to be selected
    parents =
geneticXGboost.new_parents_selection(population=population,
fitness=fitnessValue, numParents=numberOfParentsMating)

    #mate these parents to create children having parameters from
these parents (we are using uniform crossover)
    children = geneticXGboost.crossover_uniform(parents=parents,
childrenSize=(populationSize[0] - parents.shape[0],
numberOfParameters))
```

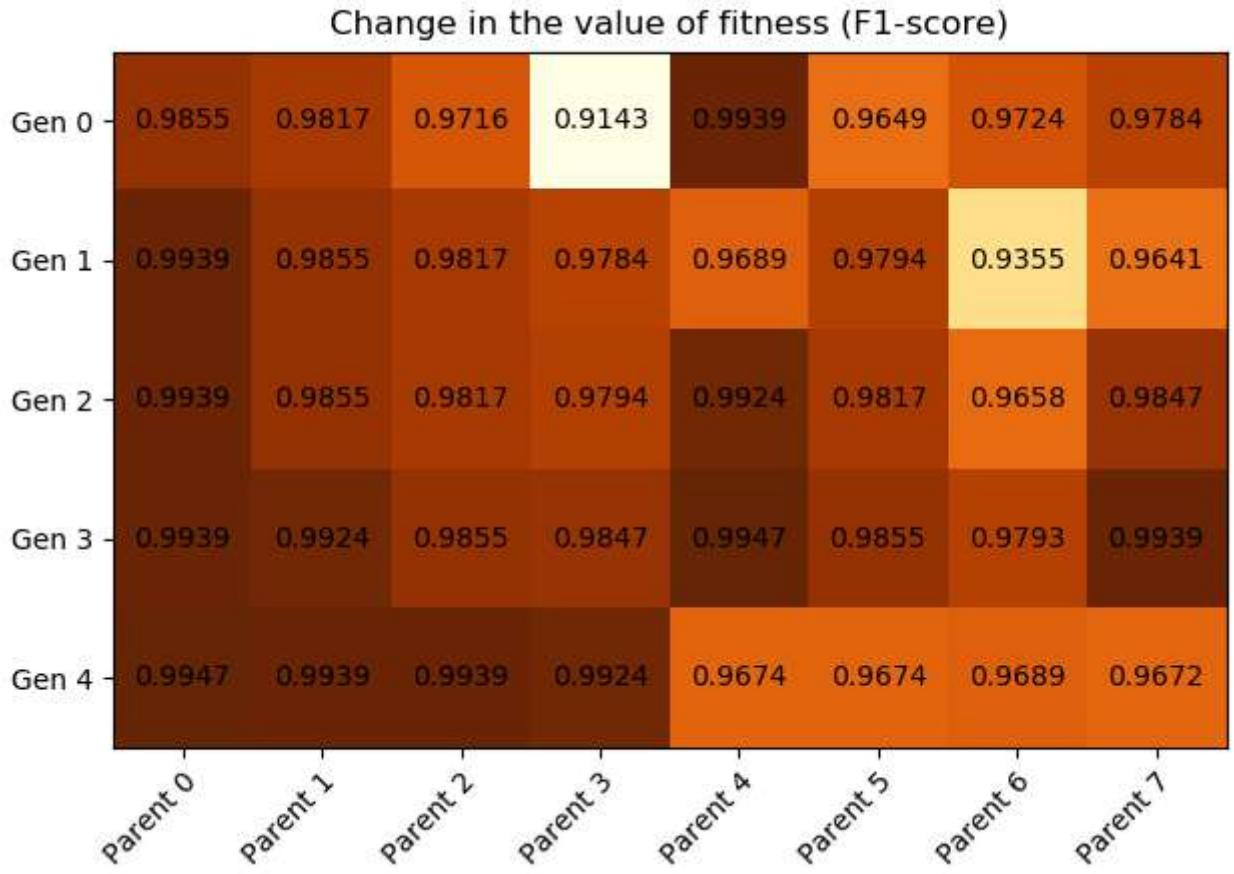
```
    #add mutation to create genetic diversity
    children_mutated = geneticXGboost.mutation(children,
numberOfParameters)

    '''
    We will create new population, which will contain parents that
where selected previously based on the
    fitness score and rest of them  will be children
    '''
    population[0:parents.shape[0], :] = parents #fittest parents
    population[parents.shape[0]:, :] = children_mutated #children

    populationHistory[(generation+1)*numberOfParents :
(generation+1)*numberOfParents+ numberOfParents , :] = population
#srore parent information
```

Finally, we get the best score and associated parameters:

```
#Best solution from the final iteration

fitness = geneticXGboost.train_population(population=population,
dMatrixTrain=xgDMatrix, dMatrixtest=xgbDMatrixTest, y_test=y_test)
fitnessHistory[generation+1, :] = fitness

#index of the best solution
bestFitnessIndex = np.where(fitness == np.max(fitness))[0][0]

#Best fitness
print("Best fitness is =", fitness[bestFitnessIndex])

#Best parameters
print("Best parameters are:")
print('learning_rate', population[bestFitnessIndex][0])
print('n_estimators', population[bestFitnessIndex][1])
print('max_depth', int(population[bestFitnessIndex][2]))
print('min_child_weight', population[bestFitnessIndex][3])
print('gamma', population[bestFitnessIndex][4])
print('subsample', population[bestFitnessIndex][5])
print('colsample_bytree', population[bestFitnessIndex][6])
```

Now let us visualize the change in fitness in the population with each generation (figure below). While we already started with high F1-score (~0.98), in two of the parents, in the randomly generated initial population, we were able to improve it further in the final generation. The lowest F1-score was 0.9143 for one parent in the initial population and the best score was 0.9947 for one of the parents in the final generation. This demonstrate that we can improve the performance metric in XGBoost, by simple implementation of genetic algorithm. The final code can be found at my

github account. It also contains code that will allow you to observe the change in various parameters, with each generation.



)

Machine Learning Xgboost Data Science Genetic Algorithm Optimization