Genetic Algorithm

2019010911 김선중

January 11, 2022

Contents

1 Introduction

Consider a situation in which an examination is taken. (Figure 1) The test consists of 20 problems and each problem is multiple choice with four answers. If we chose all the answer arbitrarily, the probability of getting the perfect score would be $\left(\frac{1}{4}\right)^{20} \approx 8.2718061 \times 10^{-25}$, which is

$$0.000000000000000000000082718061255302767487140869206996285356581211090087890625\%.$$

Figure 1: A multiple choice test

In this article, we implement the genetic algorithm to find the right answer. To sum up in advance, 40 generations of population size 200 with crossover and mutatation usually(939 out of 1000) get the full mark in the test.

I used python, especially jupyter notebook for the implementations.

2 Structure of the Genetic Algorithm

An answer to the examination can be thought of as a 20 dimensional vector, each component is among 1, 2, 3, 4. Said differently, if a is an answer, then

$$a \in \{1, 2, 3, 4\}^{20}$$

The *right answer* to the examination is also an element of $\{1, 2, 3, 4\}^{20}$. We generate the right answer by the following code; (Confer the comments after #.)

```
n_cho = 4 # the number of choices in a problem
n_pro = 20 # the number of problems in a test
right_answer = randint(1, 1 + n_cho, n_pro)
```

This produces an 20-dimensional vector and we would regard it as the right answer.

Next, we generate a set of multiple answers; With the code below, we get 50 answers.

```
n_ind = 100 ### the number of individuals in a population
pop = [randint(1, 1 + n_cho, n_pro).tolist() for _ in range(n_ind)]
```

(The symbol ###, other than # for comment means that we treat this parameter as an independent variable.) We call these answers as *individuals* and call the collection of individuals as *population* or a *generation*. What we've done is to have made the first generation.

Each individual is a candidate for the right answer. To measure the performance of the individual, we define the *fitness* function ;

```
def count(tuple1,tuple2):
    return sum([int(tuple1[i] == tuple2[i]) for i in range(len(tuple1))])
```

This `count` function literally counts the matching components between two vectors. So, to be more precise,

$$\texttt{count}(\cdot,\texttt{right\_answer}) : \{1,2,3,4\}^{20} \to \{0,1,2,\cdots,20\}$$

is the fitness function that maps an individual to the score of the individual. Using the fitness function, we can record the score of the individual in `pop` as the variable `scores`;

```
scores = [count(right_answer,c) for c in pop]
```

Thus, the `scores` consists of 50 numbers, each ranging from 0 to 20.

To produce the next generation, we select, pair up, cross-breed, and make the individuals mutate. In this article, we implemented two different types of selection.

```
n_can = 3 # the number of candidates among which one selects in select1
def select1(pop, scores, k=n_can): # choose k number of individuals randomly and select
    the fittest
  selection_ix = randint(len(pop))
  for ix in randint(0, len(pop), k-1):
    if scores[ix] > scores[selection_ix]:
      selection_ix = ix
  return pop[selection_ix]
```

Recall that `pop`, or the first generation, consists of 50 individuals. We sample three individuals from the first generation, and choose the fittest one whose score is the highest. This is the first selection strategy.

```
def select2(pop, scores): # select one individual with probability proportional to the
    score
  return np.array(random.choices(pop,weights=scores)).flatten().tolist()
```

The second strategy collects one individual from the first generation, not arbitrarily, but following the specific probability distribution. The probability of each individual are taken to be proportional to the score of it.

After implementing `select1` or `select2` multiple(=50) times to the first population and its scores, we've *selected* 50 individuals who will be *parents* of the next generation. Every two neighboring individuals pair up, say, individuals with odd order are male and ones with even order are female. Recall that each individual is a string of numbers. We pick arbitrary point of the string, concate left string of the male individual with right string of the female individual and vice versa. Then, each pair produces two vectors with same dimensions. To give more diversity to genes or the vector, we substitute each component with its complement (additive inverse ; $1 \mapsto 4$, $2 \mapsto 3$, $3 \mapsto 2$, $4 \mapsto 1$) with a (small) given probability. This procedure is called *mutation*. The vectors after mutation procedures are called *children* of the parents.

The below code is an illustration of pairing, crossover(cross-breeding) and mutation;

```
select=[_,select1,select2]
sel = 1 ### whether to use selection1 or selection2
r_cro = .9 ### the rate of crossover
r_mut = 0.05 ### the rate of mutation

def crossover(p1, p2, r_cro): # p : parent
   c1, c2 = p1.copy(), p2.copy()
   if rand() < r_cro:
      pt = randint(1, len(p1)-2)
      c1 = p1[:pt] + p2[pt:]
      c2 = p2[:pt] + p1[pt:]
   return [c1, c2] # c : child

def mutation(tuple, r_mut):
    for i in range(len(tuple)):
        if rand() < r_mut:
            tuple[i] = (5 - tuple[i]) % 5

pop = [randint(1, 1 + n_cho, n_pro).tolist() for _ in range(n_ind)]
scores = [count(right_answer,c) for c in pop]
selected = [select[sel](pop, scores) for _ in range(n_ind)]
children = list()
for i in range(0, n_ind, 2):
    p1, p2 = selected[i], selected[i+1]
    for c in crossover(p1, p2, r_cro):
        mutation(c,r_mut)
        children.append(c)
pop = children
```

This produces the second generation from the old one. We can do this iteratively, yielding multiple(say 40) generations. The code illustrated in this section is for explanatory use.

3 Experiment : Independent and Dependent Variables

By recording the average score(=`y_avg`) or the maximum score(=`y_max`) of the individuals in each generation, we can plot the trend of them. We regard the graph of `y_avg` or `y_max`, or the final value of `y_max` as the dependent varaibles.

Meanwhile, we can think of various parameters as independent variable. Here is a list of independent variables with candidate numbers with the default number (bold).

- `sel` : selction strategies **1**, 2
- `n_ind` : the number of individuals 10, 20, **50**, 100, 200
- `r_cro` : the rate of crossover 0, 0.5, **0.9**, 1
- `r_mut` : the rate of mutation 0, 0.01, **0.05**, 0.1, 0.5, 1

4 Results

4.1 Default Model

We first tried three independent experiments in the default settings (Figure 2).

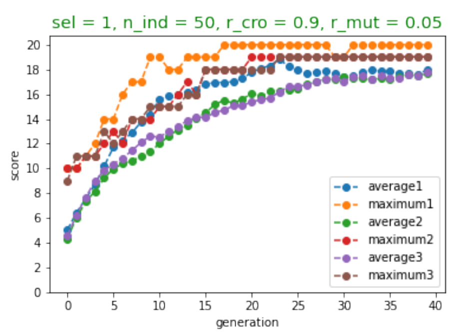


Figure 2: The first three sets of values of `y_avg` and `y_max` for the default model.

To describe many results in one graph, we conduct ten implementations and average over each generation (Figure 3). Unless otherwise specified from now on, we present graphs in this form of Figure 3. Notice that the values of parameters are presented at the top of the graph.

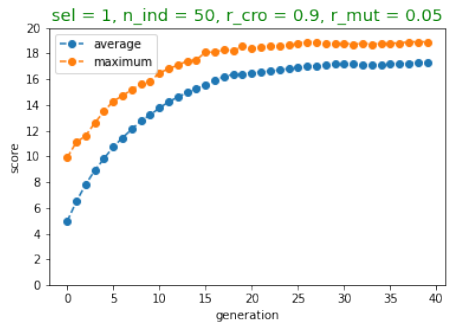


Figure 3: The average values of `y_avg` and `y_max` for the default model.

4.2 Selection Strategies

Now, we compare two different selection strategies. Figure 4, in comparison with the Figure 2, shows that the maximum scores of `selection2` fluctuate as generation passes by. That is, `y_max` doesn't necessarily increase monotonically. Moreover, the overall performance of `select2` is no better than that of `select1` (Figure 5).

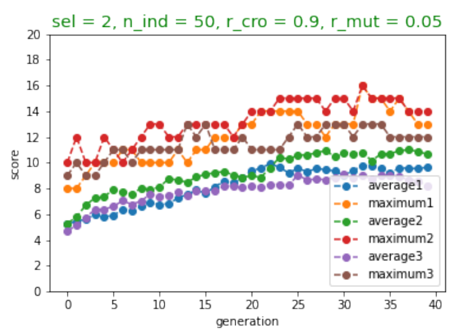


Figure 4: The first three sets of values of `y_avg` and `y_max` for the select2 model.

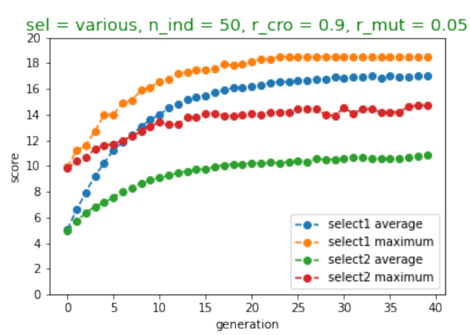


Figure 5: The comparison between `select1` and `select2`

4.3 The Number of Individuals

In the following three subsections, we consider changing parameters : the number of individuals, the rate of crossover and the rate of mutation. In these subsections, we consider only averages rather than maximums Figure 6 shows that the larger the number of individuals, the better the performance;

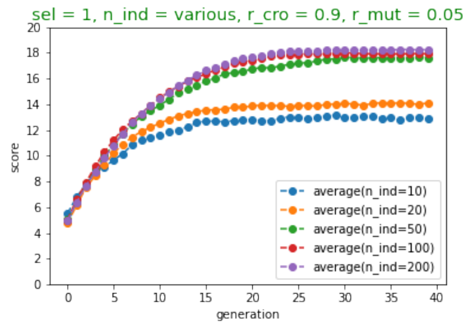


Figure 6: the average values of `y_avg` for different numbers of individuals

4.4 The Rate of Crossover

Figure 7 shows that the optimal value of `r_cro` is around 0.9. The condition `r_cro`=0.9 is better than, but not that much better than the condition `r_cro`=1.

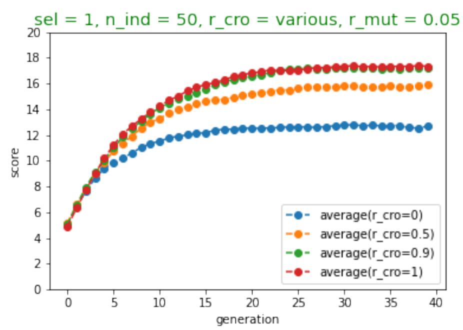


Figure 7: the average values of `y_avg` for different rates of crossover

4.5 The Rate of Mutation

Figure 8 shows the performances that result from the various rates of mutation. The optimal value among them is `r_mut`=0.01. Note that the trend of the `y_avg` for `r_mut`=1 increases in a zigzag shape. The performance becomes better as generation goes by because of the selection and the randomness of cross-breeding, but the score alternates its value because of the 100% probability of mutation.

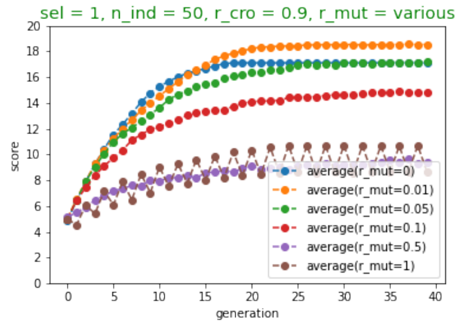


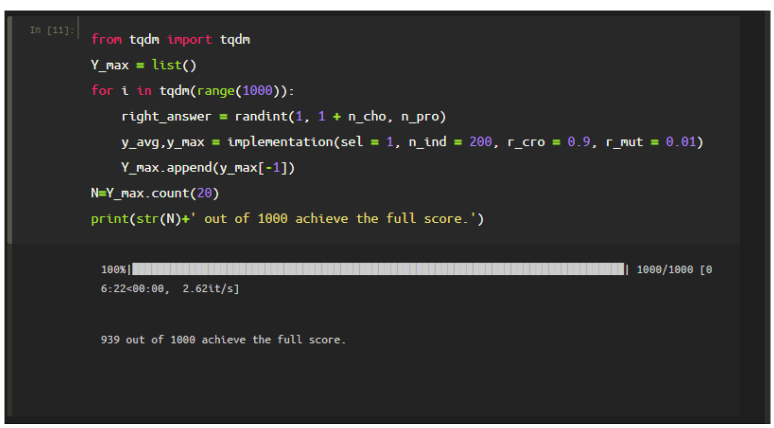
Figure 8: the average values of `y_avg` for different rates of mutation

5 Conclusions

We have applied the genetic algorithm to the problem of selecting the right answer to a multiple choice test. The crossover and mutation proved to be helpful to find better answers as is discussed in **4.4** and **4.5**. Unsurprisingly, bigger size in the population make the algorithm more accurate.

Two types of selection are experimented. The first strategy selects arbitrary three individuals and choose the best one, while the second strategy select one individual according to how good the score of the individual is. The first one proved to be better than the second one where the latter fluctuate in its performance.

I'll finish this article by presenting the best model results. With the best parameters ever considered, the genetic algorithm arrived at the right answer of (statistical) probability 93.9%;



```
In [11]:
from tqdm import tqdm
Y_max = list()
for i in tqdm(range(1000)):
    right_answer = randint(1, 1 + n_cho, n_pro)
    y_avg,y_max = implementation(sel = 1, n_ind = 200, r_cro = 0.9, r_mut = 0.01)
    Y_max.append(y_max[-1])
N=Y_max.count(20)
print(str(N)+' out of 1000 achieve the full score.')

100%|██████████████████████████████████████████| 1000/1000 [0
6:22<00:00,  2.62it/s]

939 out of 1000 achieve the full score.
```

6 Appendix : python code

```
import random
from numpy.random import randint
from numpy.random import rand
import matplotlib.pyplot as plt
```

```python
import numpy as np

# Hyperparameters
n_cho = 4 # the number of choices in a problem
n_pro = 20 # the number of problems in a test
n_ind = 50 ### the number of individuals in a population
n_gen = 40 # the number of generations
n_can = 3 # the number of candidate which one selects upon in select1
r_cro = .9 ### the rate of crossover
r_mut = 0.05 ### the rate of mutation
sel = 1 ### whether to use selection1 or selection2

# Helper functions
def count(tuple1,tuple2):
    return sum([int(tuple1[i] == tuple2[i]) for i in range(len(tuple1))])

def select1(pop, scores, k=n_can): # choose k number of individuals randomly and select
    the fittest
   selection_ix = randint(len(pop))
   for ix in randint(0, len(pop), k-1):
      if scores[ix] > scores[selection_ix]:
         selection_ix = ix
   return pop[selection_ix]

def select2(pop, scores): # select one individual with probability proportional to the
    score
   return np.array(random.choices(pop,weights=scores)).flatten().tolist()

select=[_,select1,select2]

def crossover(p1, p2, r_cross):
   c1, c2 = p1.copy(), p2.copy()
   if rand() < r_cross:
      pt = randint(1, len(p1)-2)
      c1 = p1[:pt] + p2[pt:]
      c2 = p2[:pt] + p1[pt:]
   return [c1, c2]

def mutation(tuple, r_mut):
    for i in range(len(tuple)):
        if rand() < r_mut:
            #tuple[i] = (5 - tuple[i]) % 5
            tuple[i] = 5 - tuple[i]
```

```python
def implementation(sel = 1, n_ind = 50, r_cro = .9, r_mut = 0.05):
    pop = [randint(1, 1 + n_cho, n_pro).tolist() for _ in range(n_ind)]
    y_avg, y_max = list(), list()
    for gen in range(n_gen):
        scores = [count(right_answer,c) for c in pop]
        y_avg.append(np.mean(scores))
        y_max.append(max(scores))
        #print(">generation %d : the average score is %.3f" % (gen,np.mean(scores)))
        selected = [select[sel](pop, scores) for _ in range(n_ind)]
        children = list()
        for i in range(0, n_ind, 2):
            p1, p2 = selected[i], selected[i+1]
            for c in crossover(p1, p2, r_cro):
                mutation(c,r_mut)
                children.append(c)
        pop = children
    return y_avg,y_max

for imp in range(1,4):
    right_answer = randint(1, 1 + n_cho, n_pro)
    y_avg,y_max = implementation(sel = sel, n_ind = n_ind, r_cro = r_cro, r_mut = r_mut)
    x = range(len(y_avg))
    plt.plot(x, y_avg, label = "average"+str(imp), linestyle='dashed',marker='o')
    plt.plot(x, y_max, label = "maximum"+str(imp), linestyle='dashed',marker='o')
    plt.xlabel('generation')
    plt.ylabel('score')
    plt.title('sel = '+str(sel)+', n_ind = '+str(n_ind)+', r_cro = '+str(r_cro)+', r_mut =
        '+str(r_mut), color = 'g', fontsize = 'x-large')
    plt.legend()
    plt.xticks(np.arange(0, 41, 5))
    plt.yticks(np.arange(0, 21, 2))
    plt.show
    plt.savefig('1_default_model_1')

Y_avg, Y_max = list(), list()
for imp in range(10):
    right_answer = randint(1, 1 + n_cho, n_pro)
    y_avg, y_max = implementation(sel = sel, n_ind = n_ind, r_cro = r_cro, r_mut = r_mut)
    Y_avg.append(y_avg), Y_max.append(y_max)
Y_avg = np.array(Y_avg)
Y_max = np.array(Y_max)
```

```
y_avg = np.mean(Y_avg, axis=0)
y_max = np.mean(Y_max, axis=0)
x = range(len(y_avg))
plt.plot(x, y_avg, label = "average", linestyle='dashed',marker='o')
plt.plot(x, y_max, label = "maximum", linestyle='dashed',marker='o')
plt.xlabel('generation')
plt.ylabel('score')
plt.title('sel = '+str(sel)+', n_ind = '+str(n_ind)+', r_cro = '+str(r_cro)+', r_mut =
    '+str(r_mut), color = 'g', fontsize = 'x-large')
plt.legend()
plt.xticks(np.arange(0, 41, 5))
plt.yticks(np.arange(0, 21, 2))
plt.show
plt.savefig('1_default_model_2')

sel = 2
for imp in range(1,4):
    right_answer = randint(1, 1 + n_cho, n_pro)
    y_avg,y_max = implementation(sel = sel, n_ind = n_ind, r_cro = r_cro, r_mut = r_mut)
    x = range(len(y_avg))
    plt.plot(x, y_avg, label = "average"+str(imp), linestyle='dashed',marker='o')
    plt.plot(x, y_max, label = "maximum"+str(imp), linestyle='dashed',marker='o')
    plt.xlabel('generation')
    plt.ylabel('score')
    plt.title('sel = '+str(sel)+', n_ind = '+str(n_ind)+', r_cro = '+str(r_cro)+', r_mut =
        '+str(r_mut), color = 'g', fontsize = 'x-large')
    plt.legend()
    plt.xticks(np.arange(0, 41, 5))
    plt.yticks(np.arange(0, 21, 2))
    plt.show
    plt.savefig('2_selection_strategies_1.png')

sel = 1 # '1', 2
n_ind = 50 # 10, 20, '50', 100, 200
r_cro = .9 # 0, .5, .9, 1
r_mut = 0.05 # 0, .01, .05, .1, .5, 1
right_answer = randint(1, 1 + n_cho, n_pro)

Y_avg, Y_max = list(), list()
for imp in range(10):
    right_answer = randint(1, 1 + n_cho, n_pro)
    y_avg, y_max = implementation(sel = sel, n_ind = n_ind, r_cro = r_cro, r_mut = r_mut)
    Y_avg.append(y_avg), Y_max.append(y_max)
```

```
Y_avg = np.array(Y_avg)
Y_max = np.array(Y_max)

y_avg_1 = np.mean(Y_avg, axis=0)
y_max_1 = np.mean(Y_max, axis=0)

sel = 2 # '1', 2

right_answer = randint(1, 1 + n_cho, n_pro)
y_avg, y_max = implementation(sel = sel, n_ind = n_ind, r_cro = r_cro, r_mut = r_mut)

Y_avg, Y_max = list(), list()
for imp in range(10):
    right_answer = randint(1, 1 + n_cho, n_pro)
    y_avg, y_max = implementation(sel = sel, n_ind = n_ind, r_cro = r_cro, r_mut = r_mut)
    Y_avg.append(y_avg), Y_max.append(y_max)
Y_avg = np.array(Y_avg)
Y_max = np.array(Y_max)

y_avg_2 = np.mean(Y_avg, axis=0)
y_max_2 = np.mean(Y_max, axis=0)

x = range(len(y_avg))
plt.plot(x, y_avg_1, label = "select1 average", linestyle='dashed',marker='o')
plt.plot(x, y_max_1, label = "select1 maximum", linestyle='dashed',marker='o')
plt.plot(x, y_avg_2, label = "select2 average", linestyle='dashed',marker='o')
plt.plot(x, y_max_2, label = "select2 maximum", linestyle='dashed',marker='o')
plt.xlabel('generation')
plt.ylabel('score')
plt.title('sel = various, n_ind = '+str(n_ind)+', r_cro = '+str(r_cro)+', r_mut =
    '+str(r_mut), color = 'g', fontsize = 'x-large')
plt.legend()
plt.xticks(np.arange(0, 41, 5))
plt.yticks(np.arange(0, 21, 2))
plt.show
plt.savefig('2_selection_strategies_2.png')

    sel = 1 # '1', 2
    N_ind = [10, 20, 50, 100, 200]
    r_cro = .9 # 0, .5, .9, 1
    r_mut = 0.05 # 0, .01, .05, .1, .5, 1

    Y_avg_, Y_max_ = list(), list()
```

```
    for n_ind in N_ind:
        right_answer = randint(1, 1 + n_cho, n_pro)
        Y_avg, Y_max = list(), list()
        for imp in range(10):
            right_answer = randint(1, 1 + n_cho, n_pro)
            y_avg, y_max = implementation(sel = sel, n_ind = n_ind, r_cro = r_cro, r_mut =
                r_mut)
            Y_avg.append(y_avg), Y_max.append(y_max)
        Y_avg = np.array(Y_avg)
        Y_max = np.array(Y_max)
        y_avg_ = np.mean(Y_avg, axis=0)
        y_max_ = np.mean(Y_max, axis=0)
        Y_avg_.append(y_avg_), Y_max_.append(y_max_)
    x = range(n_gen)
    for i in range(len(N_ind)):
        plt.plot(x, Y_avg_[i], label = "average(n_ind="+str(N_ind[i])+")",
            linestyle='dashed',marker='o')
        #plt.plot(x, Y_max_[i], label = "maximum(n_ind="+str(N_ind[i])+")",
            linestyle='dashed',marker='o')
    plt.xlabel('generation')
    plt.ylabel('score')
    plt.title('sel = '+str(sel)+', n_ind = various, r_cro = '+str(r_cro)+', r_mut =
        '+str(r_mut), color = 'g', fontsize = 'x-large')
    plt.legend()
    plt.xticks(np.arange(0, 41, 5))
    plt.yticks(np.arange(0, 21, 2))
    plt.show
    plt.savefig('3_n_ind.png')

sel = 1 # '1', 2
n_ind = 50 # 10, 20, '50', 100, 200
#r_cro = .9 # 0, .5, .9, 1
R_cro = [0, .5, .9, 1]
r_mut = 0.05 # 0, .01, .05, .1, .5, 1

Y_avg_, Y_max_ = list(), list()
for r_cro in R_cro:
    right_answer = randint(1, 1 + n_cho, n_pro)
    Y_avg, Y_max = list(), list()
    for imp in range(10):
        right_answer = randint(1, 1 + n_cho, n_pro)
        y_avg, y_max = implementation(sel = sel, n_ind = n_ind, r_cro = r_cro, r_mut =
            r_mut)
```

```
        Y_avg.append(y_avg), Y_max.append(y_max)
    Y_avg = np.array(Y_avg)
    Y_max = np.array(Y_max)
    y_avg_ = np.mean(Y_avg, axis=0)
    y_max_ = np.mean(Y_max, axis=0)
    Y_avg_.append(y_avg_), Y_max_.append(y_max_)
x = range(n_gen)
for i in range(len(R_cro)):
    plt.plot(x, Y_avg_[i], label = "average(r_cro="+str(R_cro[i])+")",
        linestyle='dashed',marker='o')
    #plt.plot(x, Y_max_[i], label = "maximum(n_ind="+str(N_ind[i])+")",
        linestyle='dashed',marker='o')
plt.xlabel('generation')
plt.ylabel('score')
plt.title('sel = '+str(sel)+', n_ind = '+str(n_ind)+', r_cro = various, r_mut =
    '+str(r_mut), color = 'g', fontsize = 'x-large')
plt.legend()
plt.xticks(np.arange(0, 41, 5))
plt.yticks(np.arange(0, 21, 2))
plt.show
plt.savefig('4_r_cro.png')

### sel = 1 # '1', 2
n_ind = 50 # 10, 20, '50', 100, 200
r_cro = .9 # 0, .5, .9, 1
R_mut = [0, .01, .05, .1, .5, 1]
#r_mut = 0.05 # 0, .01, .05, .1, .5, 1

Y_avg_, Y_max_ = list(), list()
for r_mut in R_mut:
    right_answer = randint(1, 1 + n_cho, n_pro)
    Y_avg, Y_max = list(), list()
    for imp in range(10):
        right_answer = randint(1, 1 + n_cho, n_pro)
        y_avg, y_max = implementation(sel = sel, n_ind = n_ind, r_cro = r_cro, r_mut =
            r_mut)
        Y_avg.append(y_avg), Y_max.append(y_max)
    Y_avg = np.array(Y_avg)
    Y_max = np.array(Y_max)
    y_avg_ = np.mean(Y_avg, axis=0)
    y_max_ = np.mean(Y_max, axis=0)
    Y_avg_.append(y_avg_), Y_max_.append(y_max_)
x = range(n_gen)
```

```
for i in range(len(R_mut)):
    plt.plot(x, Y_avg_[i], label = "average(r_mut="+str(R_mut[i])+")",
        linestyle='dashed',marker='o')
plt.xlabel('generation')
plt.ylabel('score')
plt.title('sel = '+str(sel)+', n_ind = '+str(n_ind)+', r_cro = '+str(r_cro)+', r_mut =
    various', color = 'g', fontsize = 'x-large')
plt.legend()
plt.xticks(np.arange(0, 41, 5))
plt.yticks(np.arange(0, 21, 2))
plt.show
plt.savefig('5_r_mut.png')

from tqdm import tqdm
Y_max = list()
for i in tqdm(range(1000)):
    right_answer = randint(1, 1 + n_cho, n_pro)
    y_avg,y_max = implementation(sel = 1, n_ind = 200, r_cro = 0.9, r_mut = 0.01)
    Y_max.append(y_max[-1])
N=Y_max.count(20)
print(str(N)+' out of 1000 achieve the full score.')
```

References

[1] "Simple Genetic Algorithm From Scratch in Python", Jason Brownlee, 2021,
https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/

[2] "Continuous Genetic Algorithm From Scratch With Python", Cahit bartu yazıcı, 2019,
https://towardsdatascience.com/continuous-genetic-algorithm-from-scratch-with-python-ff29deec