Genetic Algorithms Term Project

Topic: Solving Knapsack problem using genetic algorithms and using it for feature selection in Logistic Regression.

Nikunj Kumar Madhogaria

19IM10041

Satyam Gupta

19IM10027

Abstract

In this paper, we have solved the famous **Knapsack Problem using genetic algorithms**. Knapsack Problem involves selecting the most valuable items to fit in a Fixed-size/weight knapsack. Each of the given items has a size/weight and a value.

We have then shown an **application** of this **in Feature Selection for Logistic Regression.**

Goals

1. To solve the Knapsack Problem using genetic algorithms.
2. To apply our solution to Knapsack Problem in Feature Selection for Logistic Regression.

Introduction

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name "knapsack problem" dates back to the early works of the mathematician **Tobias Dantzig** (1884–1956), and refers to the commonplace problem of packing the most valuable or valuable items without overloading the luggage.

The knapsack problem is a problem in **combinatorial optimization**: Given a set of items, each with a weight/size and a value, determine the number of each item to include in a collection so that the total weight/size is less than or equal to a given limit and the total value is as large as possible.

The most common problem being solved is the **0-1 knapsack problem**, which restricts the number x_i of copies of each kind of item to zero or one. Given a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, along with a maximum weight capacity W:

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0,1\}$$

Older Methods

There are two popular methods to solve Knapsack problems:

First is the **naive method,** which compares all the 2^N permutations. As we can see, the time taken increases exponentially with an increase in N. So, this works fine with N as small as 20. If we increase N any further, it will take a lot of time, even days for a decent system if we increase N to 1000.

Second is **Dynamic Programming(DP).** The time complexity of solving it using DP is of the order of N*W, where N is the total number of items out of which we need to select, and W is the weight/size capacity. It will work only when N*W is not very large, and all the weights are Integers; dp fails with fractional weight sizes.

Therefore, we need something better for fractional weights and large numbers of items. It is tempting, therefore, to use search

heuristics like Genetic Algorithms.

Logistic Regression

Logistic regression is a supervised learning classification algorithm used to **predict the probability** of a target variable. The nature of the target or dependent variable is dichotomous, which means there would be only two possible classes.

In simple words, the dependent variable is **binary** having data coded as 1 for success/yes and 0 for failure/no.

Mathematically, a logistic regression model predicts P(Y=1) as a function of X. It is one of the simplest ML algorithms used for various classification problems such as spam detection, Diabetes prediction, cancer detection, etc.

To get the predicted output, we feed a set of features to this model. But we should **avoid using all the features** as they might contain some bad or irrelevant features that might decrease the predictions' accuracy. It is similar to the knapsack problem. In the knapsack problem, we have to choose the best items, which increases the total value, and here we are selecting the best features which are **increasing the accuracy** of the model.

Proposed Algorithm

Knapsack Problem is about **maximizing the total value** of the list of items selected while inside the Weight/Size limit. Since the weights/sizes of objects cannot be negative, our algorithm starts with the best fitness of zero, implying that the bag is empty at the start. When we get a valid combination of items with the sum of values more than the current best fitness, we update the best fitness with this number. This process is repeated until the best fitness remains constant for a fixed number of generations or the maximum number of generations is reached.

1. Creating Population

The Population in our case consists of **binary arrays** of length n. 0 implies that the element will not be selected while 1 means that it will be selected.

2. Fitness

1. **Knapsack:** Our fitness function simply takes an individual and returns the sum of its value by multiplying the element with the value of that item if its size/weight falls within the limit, else it returns -1.

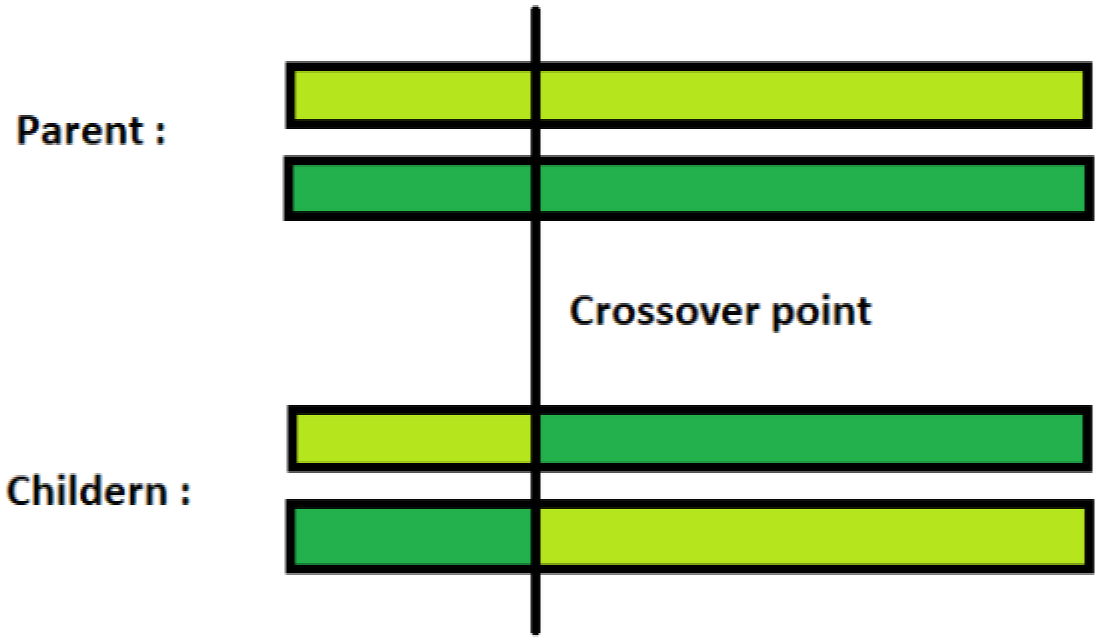
2. **Feature selection of logistic regression:** Our fitness function takes a binary array as an input and trains a logistic regression model using the features marked 1 in the array. It returns the accuracy of the predicted outcomes by **running the model on the test data.**

3. Selection

We have used **tournament selection** to solve this problem. In tournament selection, The population is first shuffled. We group individuals in pairs (tournament size = 2), and the fitter one goes to the next generation. If our population size is N, N/2 individuals have been selected so far. We repeat this process, thereby getting N/2 more individuals, thus maintaining the population size.

4. Crossover

Crossover is a genetic operator used to combine the genetic information of two parents to generate new offspring. It is one way to generate new solutions from an existing population stochastically and is analogous to the crossover in sexual reproduction in biology.



As shown in the picture above, we use a **single-point crossover** where two parents produce two offsprings in one operation. Herein we generated a random number (x) between 2 and n-2 (including one and n might result in one child is the same as one of the parents and the other child the same as the other parent) and generated two children. One has the first x elements same as the first parent and rest same as the second parent. Another child has the first n-x features the same as the first parent and the remaining elements the same as the other.

5. Mutation

The mutation is a small perturbation added to the population, helping **solutions stuck in local optima** to mutate out of it. It is used to maintain genetic diversity from one generation to another. It is performed quite sparingly. Thus, it makes sense to perform it with a small probability, usually around 5-10%. In our algorithm, we decided to use **Adaptive Mutation**. In genetic algorithms, mutation probability is generally assigned a value irrespective of their fitness value. For high fitness individuals, which have fitness **above the average fitness** of the whole population, we decided to keep the mutation probability less so that the population retains high fitness individuals after a mutation cycle. On the other hand, for lower fitness individuals, who have fitness less than the whole population's average fitness, we kept the mutation probability higher of mutating to become a better individual.

Experimental Results

1. Knapsack

We ran our code for the following input:

n=17 (Number of items)

value = [60,100,120,80,70,60,110,75,40,90,100,150,90,145,75,100,50]

weight =[10,20,30,40,50,20,35,25,10,20,60,40,20,50,50,10,50,20]

W = 150 (Max. Capacity)

We used a population size of 100 and ran the algorithm for 150 generations.

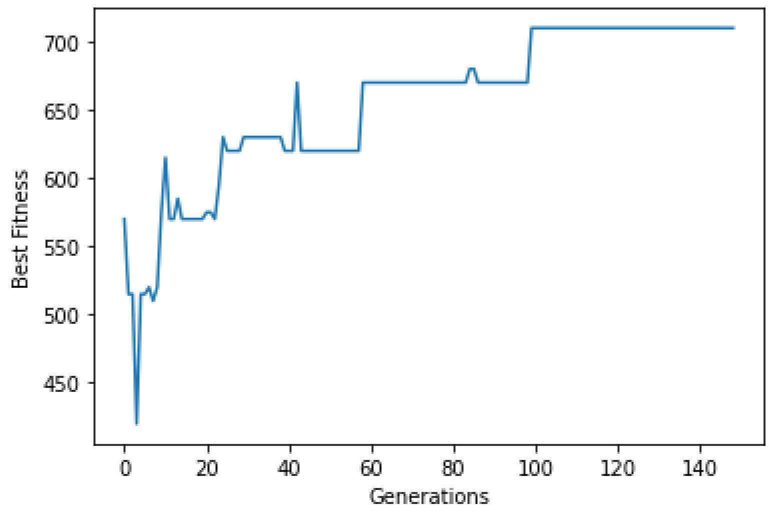
Output (Fittest individual for every 10 generations):

```
Generation:  10 Best Fitness:  520 Individual:  [1, 1, 0, 0, 0, 1, 1, 0,
0, 0, 0, 0, 1, 0, 0, 1, 0]
```

```
Generation:  20 Best Fitness:  570 Individual:  [1, 1, 0, 0, 0, 1, 1,
0, 0, 0, 0, 1, 1, 0, 0, 0, 0]
Generation:  30 Best Fitness:  620 Individual:  [1, 0, 1, 0, 0, 1, 0, 0,
1, 0, 0, 1, 1, 0, 0, 1, 0]
Generation:  40 Best Fitness:  630 Individual:  [1, 0, 1, 0, 0, 0, 1, 0,
0, 0, 0, 1, 1, 0, 0, 1, 0]
Generation:  50 Best Fitness:  620 Individual:  [1, 0, 1, 0, 0, 1, 0, 0,
1, 0, 0, 1, 1, 0, 0, 1, 0]
Generation:  60 Best Fitness:  670 Individual:  [1, 0, 1, 0, 0, 1, 0, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0]
Generation:  70 Best Fitness:  670 Individual:  [1, 0, 1, 0, 0, 1, 0, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0]
Generation:  80 Best Fitness:  670 Individual:  [1, 0, 1, 0, 0, 1, 0, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0]
Generation:  90 Best Fitness:  670 Individual:  [1, 1, 1, 0, 0, 1, 0, 0,
0, 0, 0, 1, 1, 0, 0, 1, 0]
Generation:  100 Best Fitness:  670 Individual:  [1, 0, 1, 0, 0, 1, 0, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0]
Generation:  110 Best Fitness:  710 Individual:  [1, 1, 1, 0, 0, 0, 0, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0]
Generation:  120 Best Fitness:  710 Individual:  [1, 1, 1, 0, 0, 0, 0, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0]
Generation:  130 Best Fitness:  710 Individual:  [1, 1, 1, 0, 0, 0, 0, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0]
Generation:  140 Best Fitness:  710 Individual:  [1, 1, 1, 0, 0, 0, 0, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0]
Generation:  150 Best Fitness:  710 Individual:  [1, 1, 1, 0, 0, 0, 0, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0]
```

The algorithm found the best fitness of 710 **after 110 generations.** Initially, we observed a perturbation in the population. Later, it was **stuck in a local optima** of 670, but it soon mutated out of it to reach the global optima of 710.

Here is the graph of the observations we got:



2. Feature Selection of Logistic Regression

We have used the dataset sent along with this pdf to test our code.

After cleaning, the data looks like this.

	id	acousticness	danceability	energy	explicit	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	valence	duration-min
0	2015	0.9490	0.2350	0.0276	0	0.927000	5	0.513	-27.398	1	0.0381	110.838	0.03980	3.0
1	15901	0.8550	0.4560	0.4850	0	0.088400	4	0.151	-10.046	1	0.0437	152.066	0.85900	2.4
2	9002	0.8270	0.4950	0.4990	0	0.000000	0	0.401	-8.009	0	0.0474	108.004	0.70900	2.6
3	6734	0.6540	0.6430	0.4690	0	0.108000	7	0.218	-15.917	1	0.0368	83.636	0.96400	2.4
4	15563	0.7380	0.7050	0.3110	0	0.000000	5	0.322	-12.344	1	0.0488	117.260	0.78500	3.4
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
12222	15343	0.0408	0.8090	0.8010	0	0.000000	1	0.353	-5.461	1	0.4070	81.940	0.74400	3.4
12223	1701	0.9120	0.4510	0.2400	0	0.000002	1	0.175	-14.014	1	0.0351	134.009	0.70100	2.0
12224	3351	0.3280	0.5510	0.5640	0	0.002950	2	0.352	-9.298	0	0.0338	124.883	0.89000	2.5
12225	8879	0.1220	0.0608	0.9390	0	0.991000	1	0.912	-26.324	1	0.1180	73.234	0.00558	3.1
12226	9711	0.0380	0.3890	0.7680	1	0.000000	1	0.119	-4.765	1	0.2560	90.146	0.33400	3.1

2227 rows × 18 columns

(All 18 columns cannot be shown due to space constraints).

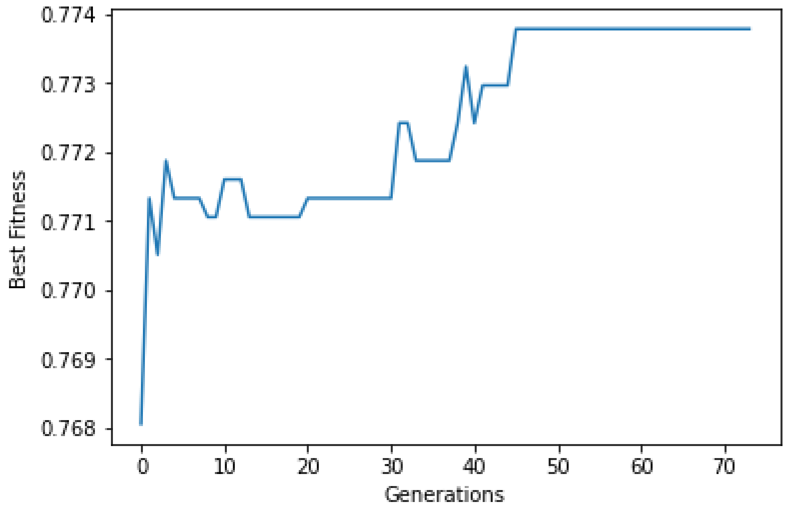
We used a population size of 50 and ran the algorithm for 70 generations.

Output (Fittest individual for every 10 generations):

```
Generation:  1 Best Fitness:  0.7694194603434178 Individual:  [1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Generation:  10 Best Fitness:  0.7710547833197057 Individual:  [1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1]

Generation:  20 Best Fitness:  0.7710547833197057 Individual:  [1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1]

Generation:  30 Best Fitness:  0.771327337149087 Individual:  [1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]

Generation:  40 Best Fitness:  0.7724175524666121 Individual:  [0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1]

Generation:  50 Best Fitness:  0.7737803216135186 Individual:  [0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1]

Generation:  60 Best Fitness:  0.7737803216135186 Individual:  [0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1]

Generation:  70 Best Fitness:  0.7737803216135186 Individual:  [0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1]
```

The algorithm found that the best **predicted accuracy is 77.378%**. Initially, it was **stuck in local optima of 77.105%**. After around 15 generations, it mutated out of it, and after some more generations, the algorithm converged to 77.378%.

Here is the graph of the observations we got:



Application

Knowingly or unknowingly, we solve many Knapsack Problems in our everyday life. Whenever we have load or size constraints, and we try to select some out of n things to include at that point in time to add maximum value to whatever we are doing.

We have stated here, few of the real-life applications of the Knapsack Problem:

1. Resource Allocation with Financial Constraints

   The problem often arises in resource allocation. The decision-makers have to **choose from a set of non-divisible projects or tasks** under a fixed budget or time constraint, respectively. There might be n machines that, if we run, cost us different amounts of money, and upon selling all the products manufactured from the machines, we earn different revenues.

Hence, to find out the best set of machines to run for a particular budget constraint, we can solve a Knapsack Problem using their costs as weight and revenues as values.

2. Construction and Scoring of Heterogeneous Test

In the construction and scoring of tests in which the test-takers have a choice as to which questions they answer. For small examples, it is a fairly simple process to provide the test-takers with such a choice. For example, if an exam contains 12 questions each worth 10 points, the test-taker need only answer 10 questions to achieve a maximum possible score of 100 points. However, on tests with a **heterogeneous distribution of point values**, it is more difficult to provide choices. **Feuerman and Weiss** proposed a system in which students are given a heterogeneous test with a total of 125 possible points. The students are asked to answer all of the questions to the best of their abilities. Of the possible subsets of problems whose total point values add up to 100, a Knapsack Algorithm would determine which subset gives each student the highest possible score.

3. TestSelection of Capital Investments

When having **multiple options to invest** a fixed amount of money, we can solve a Knapsack Problem to find out the best set of Investments that will be most profitable to us by using their costs as weights and profits as values.

Conclusion

Genetic Algorithms are essentially **computer simulations of nature**. The fitness of the population tends to improve with iterations. The ***survival of the fittest*** paradigm is reflected in the Selection operator. Crossover resembles reproduction, just like parents giving birth to offspring who have genetic information from both parents. The algorithm's sublime simplicity and the impressive results show that it produces to show how powerful these algorithms are.

We had a great time trying to solve a **real-life problem using genetic algorithms**. Only upon implementing the algorithm could we feel why these algorithms are known as *Genetic Algorithms* and their resemblance to **Darwin's Theory of Evolution**.

References

1. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.
2. Yu and M. Gen, *Introduction to Evolutionary Algorithms*, ser. Decision Engineering. Springer, 2010.
3. Multi-Objective Optimization Using Evolutionary Algorithms by Kalyanmoy Deb.

Code of Knapsack:

```
import random
import matplotlib.pyplot as plt
import numpy as np
from array import *

n=17
value = [60,100,120,80,70,60,110,75,40,90,100,150,90,145,75,100,50]
weight =[10,20,30,40,50,20,35,25,10,20,60,40,20,50,50,10,50,20]
W = 150
def create_individual():
    individual = []
    for i in range(n):
        individual.append(random.randint(0,1))
    return individual

population_size = 100
generation = 0
population = []
for i in range(population_size):
    individual = create_individual()
    population.append(individual)

def fitness(individual):
    fitness = 0
    weight2 = 0
    for i in range(n):
        fitness+=(value[i])*(individual[i])
        weight2+=(weight[i])*(individual[i])
    if weight2 > W:
        fitness = -1
    return fitness

def crossover(parent1, parent2):
    position = random.randint(2, n-2)
    child1 = []
    child2 = []
    for i in range(position+1):
        child1.append(parent1[i])
        child2.append(parent2[i])
    for i in range(position+1, n):
        child1.append(parent2[i])
        child2.append(parent1[i])
    return child1, child2

def mutation1(individual):
    probability = 0.10
    check = random.uniform(0, 1)
    if(check <= probability):
     for i in range(n):
        check1=random.uniform(0,1)
        if(check1<=probability):
          individual[i]=1-individual[i]
    return individual
def mutation2(individual):
    probability = 0.15
    check = random.uniform(0, 1)
    if(check <= probability):
     for i in range(n):
        check1=random.uniform(0,1)
        if(check1<=probability):
          individual[i]=1-individual[i]
    return individual
```

```python

def tournament_selection(population):
    new_population = []
    for j in range(2):
        random.shuffle(population)
        for i in range(0, population_size-1, 2):
            if fitness(population[i]) > fitness(population[i+1]):
                new_population.append(population[i])
            else:
                new_population.append(population[i+1])
    return new_population

best_fitness = fitness(population[0])
fittest_individual = 0
gen = 0
answer = best_fitness
all_best=[]
while(gen!=150):
    gen +=1
    population = tournament_selection(population)
    new_population = []
    random.shuffle(population)
    for i in range(0, population_size-1, 2):
        child1, child2 = crossover(population[i], population[i+1])
        new_population.append(child1)
        new_population.append(child2)
    tot_fitness=0
    for i in new_population:
        tot_fitness+=fitness(i)
    avg_fitness=tot_fitness/population_size
    for individual in new_population:
        if(fitness(individual) < avg_fitness):
            individual = mutation1(individual)
        else:
            individual = mutation2(individual)
    population = new_population
    best_fitness = fitness(population[0])
    for individual in population:
        if fitness(individual) > best_fitness:
            best_fitness = fitness(individual)
            fittest_individual = individual
    all_best.append(best_fitness)
    answer = max (answer,best_fitness)
    if gen%10 == 0:
        print("Generation: ", gen, "Best Fitness: ", best_fitness, "Individual: ",fittest_individual)
```

Code of Feature Selection of Logistic Regression:

```
import random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.preprocessing import StandardScaler
from array import *

data = pd.read_csv('Train_data.csv')
release_date = data['release_date'].str.split('-',expand = True)
data['release_day'] = pd.to_numeric(release_date[0])
data['release_month'] = pd.to_numeric(release_date[1])
data['release_year'] = pd.to_numeric(release_date[2])
data.drop(columns =["release_date"], inplace = True)
data.drop(columns =["year"], inplace = True)
popularity=data['popularity']
popularity.replace(['very low','low'],0,inplace=True)
popularity.replace(['average','high','very high'],1,inplace=True)
data['mode'].replace(['Major'],1,inplace=True)
data['mode'].replace(['Minor'],0,inplace=True)
data['explicit'].replace(['No'],0,inplace=True)
data['explicit'].replace(['Yes'],1,inplace=True)
Y = data['popularity']
X = data.drop("popularity", axis = 1)
columns = X.columns
scaler = StandardScaler()
X_std = scaler.fit_transform(X)
X_std = pd.DataFrame(X_std, columns = columns)
x_train, x_test, y_train, y_test = train_test_split(X_std, Y, test_size = 0.15, random_state = 45)
lr_std = LogisticRegression()
lr_std.fit(x_train, y_train)
y_pred = lr_std.predict(x_test)
def create_individual():
    individual = []
    for i in range(n):
        individual.append(random.randint(0,1))
    return individual

population_size = 50
generation = 0
population = []
for i in range(population_size):
    individual = create_individual()
    population.append(individual)

def fitness(array):
    indices = []
    j=0
    for i in X_std:
        if array[j]==1:
            indices.append(i)
        j=j+1
    model=X_std[indices]

    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(model,
                                                        Y, test_size=0.30,
                                                        random_state=101)
    from sklearn.linear_model import LogisticRegression
    logmodel = LogisticRegression()
    logmodel.fit(X_train,y_train)
```

```python
    predictions = logmodel.predict(X_test)
    return logmodel.score(X_test, y_test)

def crossover(parent1, parent2):
    position = random.randint(2, n-2)
    child1 = []
    child2 = []
    for i in range(position+1):
        child1.append(parent1[i])
        child2.append(parent2[i])
    for i in range(position+1, n):
        child1.append(parent2[i])
        child2.append(parent1[i])
    return child1, child2

def mutation1(individual):
    probability = 0.10
    check = random.uniform(0, 1)
    if(check <= probability):
      for i in range(n):
        check1=random.uniform(0,1)
        if(check1<=probability):
          individual[i]=1-individual[i]
    return individual
def mutation2(individual):
    probability = 0.15
    check = random.uniform(0, 1)
    if(check <= probability):
      for i in range(n):
        check1=random.uniform(0,1)
        if(check1<=probability):
          individual[i]=1-individual[i]
    return individual

def tournament_selection(population):
    new_population = []
    for j in range(2):
        random.shuffle(population)
        for i in range(0, population_size-1, 2):
            if fitness(population[i]) > fitness(population[i+1]):
                new_population.append(population[i])
            else:
                new_population.append(population[i+1])
    return new_population

best_fitness = fitness(population[0])
fittest_individual = 0
gen = 0
answer = best_fitness
all_best=[]
while(gen!=70):
    gen += 1
    population = tournament_selection(population)
    new_population = []
    random.shuffle(population)
    for i in range(0, population_size-1, 2):
        child1, child2 = crossover(population[i], population[i+1])
        new_population.append(child1)
        new_population.append(child2)
    tot_fitness=0
    all_fitness=[]
    for i in new_population:
```

```python
    for i in new_population:
        f=fitness(i)
        tot_fitness+=f
        all_fitness.append(f)
    avg_fitness=tot_fitness/population_size
    for individual in range(population_size):
        if(all_fitness[individual] < avg_fitness):
            new_population[individual] = mutation1(new_population[individual])
        else:
            new_population[individual] = mutation2(new_population[individual])
    population = new_population
    all_fitness=[]
    for i in population:
        f=fitness(i)
        tot_fitness+=f
        all_fitness.append(f)
    avg_fitness=tot_fitness/population_size
    best_fitness = fitness(population[0])
    for individual in range(population_size):
        if all_fitness[individual] > best_fitness:
            best_fitness = all_fitness[individual]
            fittest_individual = population[individual]
    answer = max (answer,best_fitness)
    all_best.append(best_fitness)
    if gen%10 == 0:
        print("Generation: ", gen, "Best Fitness: ", best_fitness, "Individual: ",fittest_individual )
print("Best Fitness: ", best_fitness, "Best Individual: ",fittest_individual )
```

--------------------END--------------------