Optimum Assignment of Machine Problems to Technicians Using a Genetic Algorithm

Dr. C. Muthu
Associate Professor, Department of Statistics
St. Joseph's College, Tiruchirappalli
&
M. C. Prakash
PG Student, Bharathidasan University, Tiruchirappalli

Abstract

The best possible solution to a business problem shall be often found by trying many different solutions and scoring them to determine their quality. The stochastic optimization techniques, such as the Genetic Optimization Technique, shall be used for this purpose. In this paper, the optimum assignment of machine problems to experienced technicians in a large manufacturing unit is discussed. Sometimes, there may be even hundreds of machine problems in a big manufacturing unit, competing for the attention of a very few experienced technicians, to get fixed. A genetic optimization algorithm is used in this paper for obtaining the optimum assignment of machine problems to technicians.

Key Terms: Stochastic optimization, Genetic Algorithm, Optimum Assignment.

1. Introduction

The collaborative filtering techniques are now increasingly used in the field of data science [1]. The Hadoop Ecosystem now plays an important role in big dataanalytics [2]. Price Predictors are often designed by the Data Analysts by using the KNN algorithm [3]. Potential insights into the customer preferences are now obtained by the Data Analysts by applying the Hierarchical Clustering algorithms [4]. In this paper, the optimum allocation of machine problems, which are often encountered in a big manufacturing company, to a small set of experienced Technicians is obtained by using the Genetic algorithm.

We consider here ten machine problems which are vying for getting the attention of five experienced technicians. For the purpose of fixing a machine problem, the manufacturing floor supervisor may indicate a particular technician as his first choice. Sometimes, this assignment may not be possible if the particular technician has already been assigned another machine problem. So, the supervisor will also indicate another technician as his second choice for fixing every machine problem. We now create a new python file named **fixFault.py** and add the list of technicians and the list of machine problems, along with their top two choices of technicians:

```
import random
import math
# The technicians, each of whom has two available slots
# for solving two machine problems at a time
technicians = ['Jerald', 'Giftson', 'Diravium', 'Yokesh', 'Bala']
# Technicians, along with their first and second choices
preferences =    [ ('Problem1', ('Yokesh', 'Diravium')),
                   ('Problem2', ('Jerald', 'Bala')),
                   ('Problem3', ('Giftson', 'Jerald')),
                   ('Problem4', ('Jerald', 'Bala')),
                   ('Problem5', ('Giftson', 'Yokesh')),
                   ('Problem6', ('Diravium', 'Bala')),
                   ('Problem7', ('Bala', 'Giftson')),
                   ('Problem8', ('Yokesh', 'Diravium')),
                   ('Problem9', ('Yokesh', 'Diravium')),
                   ('Problem10', ('Diravium', 'Giftson'))]
```

Here, each problem cannot be assigned to a technician, who has been indicated as the *first choice* for solving this problem by the supervisor. For example, only two problems can be allotted at a time to the technician Yokesh, but in the case of Problem 1, Problem 8 and Problem 9, Yokesh is indicated as the *first choice* by the supervisor. Assigning any of these problems to the second-choice technician will mean there will not be enough slots for Diravium for the problems proposed to be assigned to him as the first choice.

The problem involved in assigning the problems to the appropriate technicians will become acute when hundreds of problems are to be assigned to the technicians in a large manufacturing unit. In such a case, it is not possible to consider all possible solutions as there may be even 1,00,000 possible solutions at times. This is so when we assume that four problems can be assigned at a time to each technician in a large manufacturing unit.

2. Cost Function

If we form a cost function that will return a very high value for invalid solutions, it will make it very difficult for the optimization algorithm to find better solutions because it has no way to determine if it is close to other good or even valid solutions. In general, it is better not to waste processor cycles searching among invalid solutions.

A better way to approach the issue is to find a way to represent solutions so that every solution is valid. A valid solution need not be necessarily a good solution. It just means that there are exactly two problems assigned to each technician. One way to do this is to think of every technician as having two slots, so that there are ten slots in total. Each problem, in order, is assigned to one of the open slots. The first problem can be placed in any one of the ten slots, the second problem can be placed in any of the nine remaining slots, and so on. The domain for searching has to capture this restriction. We have to add the following line of code to **fixFault.py:**

```
# [(0,9), (0,8), (0,7), (0,6), ..., (0,0)]
domain = [(0, (len(technicians) * 2) – i – 1) for i range
        (0, len (technicians) * 2)]
```

The **cost function** starts with the construction of a list of slots. The slots are removed as they are used up. The cost is calculated by comparing a problem’s current technician assignment to its top two choices. The total cost will not increase if the problem is currently assigned to its top choice technician, by 1 if it is assigned to its second-choice technician, and by 3 if it is not assigned to either of its choices. Let us add the following cost function to **fixFault.py**:

```
def technicianCost (vec):
    cost = 0
    # A list of slots is created
    slots = [0, 0, 1, 1, 2, 2, 3, 3, 4, 4]
    # Looping over each problem
    for i in range (len(vec)):
    × = int (vec[i])
    technician = technicians [slots[x]]
    preference = preferences[i][1]
    # First choice costs 0, second choice costs 1
    if preference[0] == technician: cost += 0
    elif preference[1] == technician: cost += 1
    else : cost += 3
    # Not on the list costs 3
    # Remove selected slot
    del slots [x]
return cost
```

3. Genetic Optimization Technique

The goal of our optimization problem is to minimize the cost by choosing the correct assignment of machine problems to the available technicians. Testing every combination will guarantee that we get the best answer, but it will take a very long time on most types of computers. Trying a few thousand random guesses and seeing which one is best is another possible technique. Randomly trying different solutions is very inefficient because it does not take advantage of the good solutions that have already been discovered.

Genetic Algorithms work by initially creating a set of random solutions known as the **population**. At each step of the optimization, the cost function for the entire population is calculated to get a ranked list of solutions. After the solutions are ranked, a new population - known as the next **generation** - is created. First, the top solutions in the current population are added to the new population as they are. This process is called **elitism**. The rest of the new population consists of completely new solutions that are created by modifying the best solutions.

There are two ways in which the solutions can be modified. The simpler of these is called **mutation**, which is usually a small, simple, random change to an existing solution. In our study, a mutation can be done simply by picking one of the numbers in the solution and increasing or decreasing it.

The other way to modify solutions is called **crossover** or **breeding**. This method involves taking two of the best solutions and combining them in some way. In this study, a simple way to do crossover is to take a random number of elements from one solution and the rest of the elements from another solution.

A new population, usually the same size as the old one, is created by randomly mutating and heading the best solutions. Then the process repeats - the new population is ranked and another population is created. This continues either for a fixed number of iterations or until there has been no improvement over several generations. We add now the **geneticOptimization( )** function to **fixFault.py**:

```
def geneticOptimization (domain, costf, populationSize = 50, step = 1,
mutationProbability = 0.2, elite = 0.2 maximumIterations = 100):
    # Mutation Operation is performed
    def mutate (vec) :
        i = random.randint (0, len (domain) – 1)
        if random.random ( ) < 0.5 and vec[i] > domain[i][0] :
            return vec[0:i] + [vec[i] – step] + vec[i+1:]
        elif vec[i] < domain[i][1] :
            return vec[0:i] + [vec[i] + step] + vec[i+1:]
        # Crossover Operation is performed
        def crossover(r1, r2):
            i = random.randint(1, len(domain) –2)
            return r1[0:i] + r2[i:]
        # The initial population is built
        population = [ ]
        for i in range (populationSize):
            vec = [random.randint(domain[i][0], domain[i][1])
                for i in range (len(domain))]
            population.append(vec)
        # To find the number of winners from each generation
        topelite = int (elite * populationSize)
        # Main loop starts here
        for i in range (maximumIterations):
```

```
        scores = [(costf (v), v) for v in population]
        scores.sort ( )
        ranked = [v for (s,v) in scores]
        # Starting with the pure winners
        population = ranked [0 : topelite]
        # Adding mutated and bred forms of the winners
        while len(population) < populationSize:
            if random.random( ) < mutationProbability:
                # Mutation is done
                c = random.randint (0, topelite)
                population.append(mutate(ranked[c]))
            else:
                # Crossover is done
                C1 = random.randint (0, topelite)
                C2 = random.randint (0, topelite)
                population.append (crossover(ranked[c1],
                                                  ranked[c2]))
        # Printing current best score
        print scores[0][0]
    return scores[0][1]
```

Once the optimum assignment of problems to the appropriate technicians is obtained with the help of the above **geneticOptimization( )** function, it shall be printed by using the following **printAssignment( )** function, which is to be added to **fixFault.py**:

```
def printAssignment(vec):
    slots = [ ]
    # Two slots are created for each technician
    for i in range (len(technicians) : slots += [i,i]
    # Looping over each problem assignment
    for i in range (len(vec)):
        x = int(vec[i])
        # Choosing the slot from the remaining ones
        technician = technicians [slots[x]]
        # Displaying the problem's name and assigned technician's name
        print preferences[i][0], technician
        # Remove this slot
        del slots[x]
```

In the following python session, we obtain the optimum assignment of problems to the technicians and display the same:

```
>>> reload (fixFault)
>>> s = fixFault.geneticOptimization(fixFault.domain, fixFault.technicianCost)
>>>fixFault.printAssignment(s)
Problem1 Giftson
Problem2 Bala
Problem3 Jerald
Problem4 Bala
problem5 Diravium
Problem6 Diravium
Problem7 Yokesh
problem8 Yokesh
Problem9 Giftson
problem10 Jerald
```

4. Conclusion

The genetic optimization algorithm is used in this paper to obtain the above stated optimum assignment of machine problems to the technicians.

References

1. Jacques Bughin, "Big Data, Big Bang?", *Journal of Big Data*, 2016, Vol.3, Iss. 2, pp. 1-14.
2. Muthu, C. and Prakash, M.C., "Impact of Hadoop Ecosystem on Big Data Analytics", *International Journal of Exclusive Management Research* - Special Issue, 2015, Vol. 1, pp. 88-90.
3. Muthu, C. and Prakash, M.C., "Building a Price Predictor for an Auctioning Website", RETELL, 2015, Vol. 15, Iss. 1, pp. 135-137.
4. Muthu, C. and Prakash, M.C., "Hierarchical Clustering of Users' Preferences", RETELL, 2016, Vol. 16, Iss. 1, pp. 135-136.
5. Muthu, C. and Prakash, M.C., "Matching the users of a Website using SVM Technique", RETELL, 2017, Vol. 17, Iss. 1, pp. 53-56.
6. Muthu, C. and Prakash, M.C., "Using Bayesian Classifier for Email Sorting", RETELL, 2017, Vol. 17, Iss. 1, pp. 57-60.