Roll no - 41307, 41310, 41315
Class :BE Comp

LP4 SCOA: Mini project No. 1

Title: Apply the Genetic Algorithm for optimization on a dataset obtained from UCI ML repository. (IRIS Dataset).

Prerequisite:

-Basic of Python, Data Mining Algorithm, Iris Dataset ,Genetic algorithm.

Software Requirements:

-Anaconda with Python 3.7

Hardware Requirement:

-PIV, 2GB RAM, 500 GB HDD, Lenovo A13-4089Model.

Learning Objectives:

-Learn How to Apply Genetic Algorithm for given Iris Dataset. The main objective of this assignment is to implement Iris Flower Dataset or any other dataset into a data frame using python .

Outcomes:

-After completion of this assignment students are able Implement code for the Iris Dataset with plotting diagram.

Theory Concepts:

1. Python is an interpreted high level programming language for general purpose programming created by Guido Van Rassom and First released in 1991.
2. Python for a design philophy that emphasizes code readability, notably using significant white space .
3. Python features a dynamic type of automatic memory management support multiple programming paradigm ,including object – oriented, imperative ,functional and procedural and has a large ,comprehensive standard

library.
4. Python library is a collection of function and methods that allows you to performs lots of actions without writing your own code.
Eg: If you are working with data, numpy , scipy, pandas ,etc .are the libraries you must know.

- **Import pandas as pd**

  Pandas is an open source ,BSD-licensed library providing high performance, easy to use data structure and data analysis tools for the python programming language.

  - sudo apt-get install python 3.6
  - sudo apt-get install python pip
  - sudo apt-get install python pandas

- **Import matplotlib as plt**

  - Matplotlib is a plotting library for the python programming language and its numerical mathematics extension numpy.
  - If provides an object oriented API for embedding plot into applications using general purpose GUI tools like Tkinter , Wxpython.

- **Iris Dataset**

  - This dataset includes three species with so samples each as well as some properties about each flower .
  - The available columns in this dataset are : id , sepal length cm, sepal width cm, petal length cm , petal width cm and species .
  - The Dataset is self available below in csv file . This dataset is also available in scikit-learn package of which the link description also attached in title.
  - The main task in this dataset is to create an iris (name of a flower) Classifier based on given properties that are the sepal and petal size.
  - If you don’t know the difference between sepal and petal, here is an image that shows which part of the flower is sepal and which part is petal.

IRIS dataset

Iris Versicolor

Iris Setosa

Iris Virginica

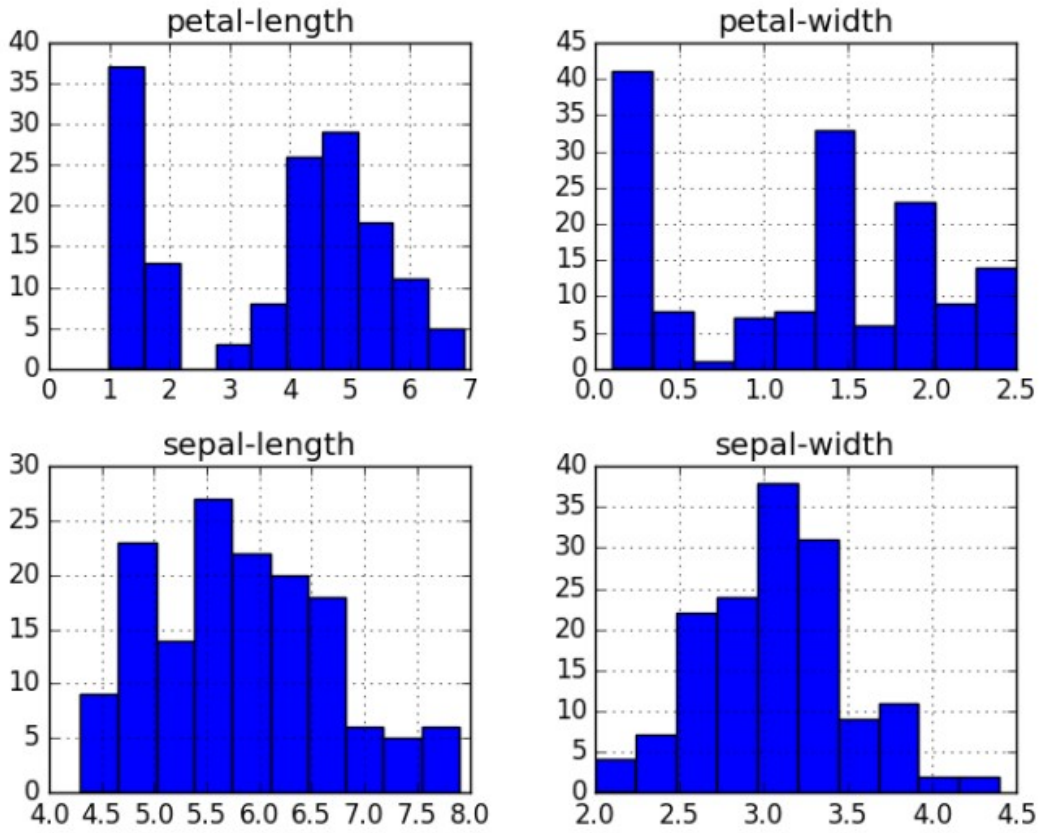
- df.isnull().any()
  - It is used to check whether we have null values in our dataset or not .

- df.types()
  - To know the type of each column values .

- df.describe()
  - check the quick summary of data.

- Df.['petalwidth'].plot.hist()
  Plt.show()

  - It is used to represent flowers datasets of values between 0.1 and 0.5 in the graph form.

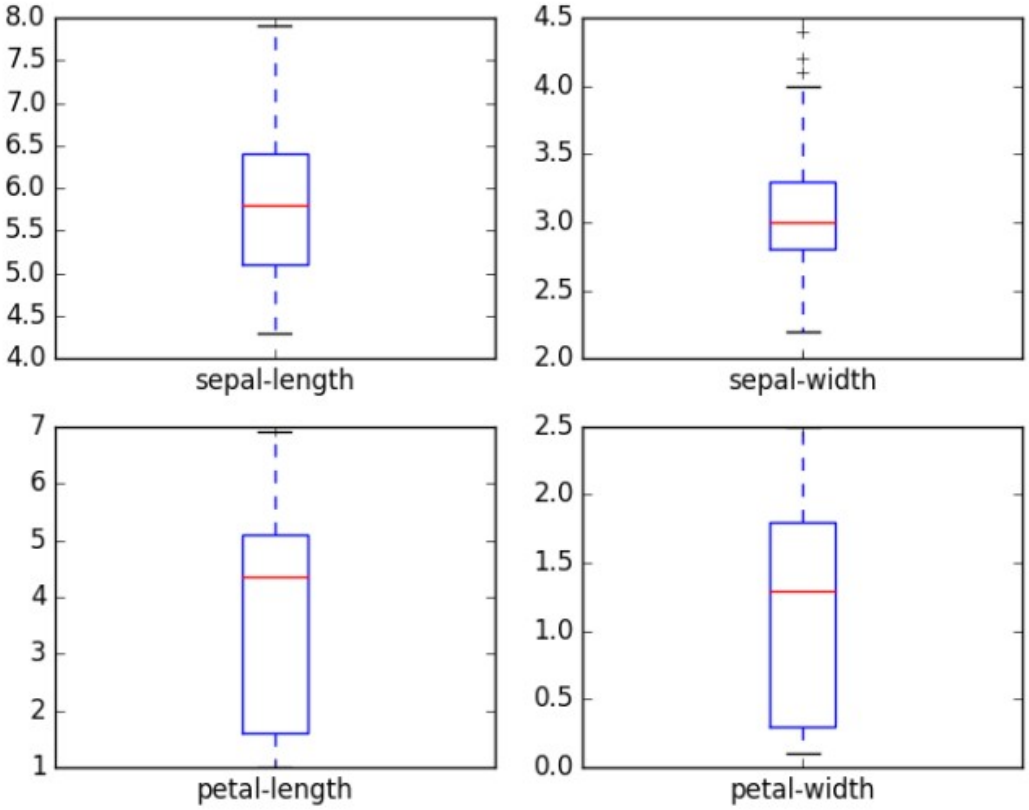


- **Splitting the Dataset**
  - Since there is only dataset available, we need to divide the dataset into training and test dataset.
  - To do this, we can use train_test_split method from the scikit learn.

- **BoxPlot**
  - A ‘Boxplot’ or ‘box-&-whiskerplot’ is a graphical summary of the distribute.
  - The box in the middle indicates ‘hinges’ and ‘median’.
  - The lines(‘whisker’)show the largest or smallest observation that falls within a distance of 1.5 times the box size from the nearest hinge.
    If any observation fall farther away, the additional points are considered ‘extreme’ values and are shown separately.
  - A boxplot can often give a good idea of the data distribution and is often more useful to compare distributions side by side as it is more compact than histogram.
  - We can use the boxplot function to calculate quick summaries for all the

variables in our dataset by default.

- The real power of boxplots is really to do comparisons of variables by sub-grouping.



Genetic Algoritm :

1. Genetic Algorithms (GAs) are search based algorithms based on the concepts of natural selection and genetics. GAs are a subset of a much larger branch of computation known as **Evolutionary Computation**.

Advantages of GAs

GAs have various advantages which have made them immensely popular. These include −

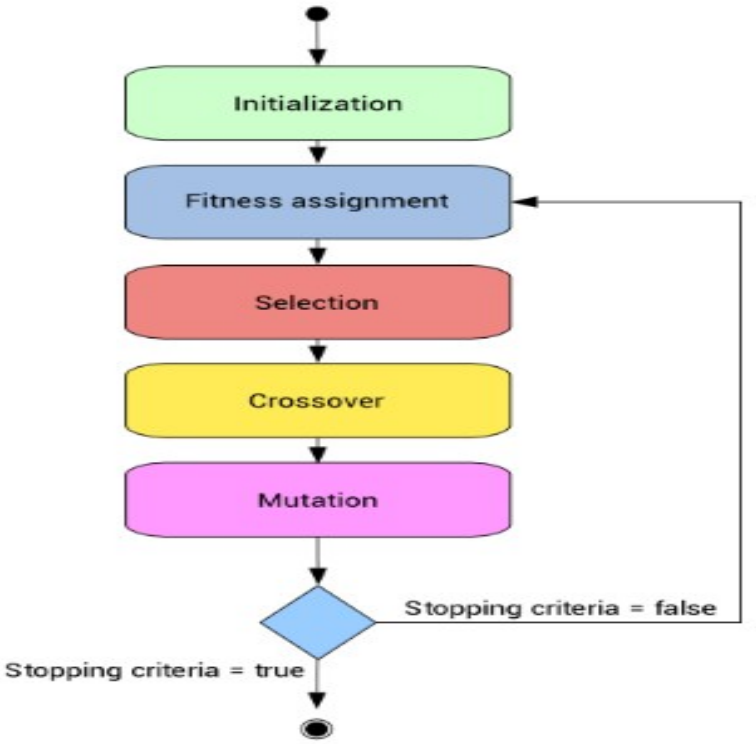
- Does not require any derivative information (which may not be available for many real-world problems).
- Is faster and more efficient as compared to the traditional methods.

- Has very good parallel capabilities.
- Optimizes both continuous and discrete functions and also multi-objective problems.
- Provides a list of “good” solutions and not just a single solution.
- Always gets an answer to the problem, which gets better over the time.
- Useful when the search space is very large and there are a large number of parameters involved.

Limitations of GAs

Like any technique, GAs also suffer from a few limitations. These include −

- GAs are not suited for all problems, especially problems which are simple and for which derivative information is available.
- Fitness value is calculated repeatedly which might be computationally expensive for some problems.
- Being stochastic, there are no guarantees on the optimality or the quality of the solution.
- If not implemented properly, the GA may not converge to the optimal solution.



Conclusion

Hence ,We have studied and practically implemented Iris flower dataset into a Data frame And we learn Genetic Algorithm for optimizing Iris Dataset .

Code :

```
import random
import numpy as np
import pandas as pd
import copy
import time
from sklearn.preprocessing import OneHotEncoder

class Network(object):

    def __init__(self, sizes):

        '''The list ``sizes`` contains the number of neurons in the
        respective layers of the network.  For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron.  The biases and weights for the
        network are initialized randomly, using a Gaussian
        distribution with mean 0, and variance 1.  Note that the first
        layer is assumed to be an input layer, and by convention we
        won't set any biases for those neurons, since biases are only
        ever used in computing the outputs from later layers.'''

        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

        # helper variables
        self.bias_nitem = sum(sizes[1:])
        self.weight_nitem = sum([self.weights[i].size for i in range(self.num_layers-2)])

    def feedforward(self, a):
        '''Return the output of the network if ``a`` is input.'''
        for b, w in zip(self.biases, self.weights):
            a = self.sigmoid(np.dot(w,a)+b)
        return a

    def sigmoid(self, z):
        '''The sigmoid function.'''
```

```
        return 1.0/(1.0+np.exp(-z))

    def score(self, X, y):

        '''
        @X = data to test
        @y = data-label to test
        @returns = score of network prediction (less is better)
        @ref: https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications
        '''

        total_score=0
        for i in range(X.shape[0]):
            predicted = self.feedforward(X[i].reshape(-1,1))
            actual = y[i].reshape(-1,1)
            total_score += np.sum(np.power(predicted-actual,2)/2)  # mean-squared error
        return total_score

    def accuracy(self, X, y):

        '''
        @X = data to test
        @y = data-label to test
        @returns = accuracy (%) (more is better)
        '''

        accuracy = 0
        for i in range(X.shape[0]):
            output = self.feedforward(X[i].reshape(-1,1))
            accuracy += int(np.argmax(output) == np.argmax(y[i]))
        return accuracy / X.shape[0] * 100

    def __str__(self):
        s = "\nBias:\n\n" + str(self.biases)
        s += "\nWeights:\n\n" + str(self.weights)
        s += "\n\n"
        return s

class NNGeneticAlgo:

    def __init__(self, n_pops, net_size, mutation_rate, crossover_rate, retain_rate, X, y):

        '''
        n_pops   = How much population do our GA need to create
        net_size = Size of neural network for population members
        mutation_rate = probability of mutating all bias & weight inside our network
        crossover_rate = probability of cross-overing all bias & weight inside out network
        retain_rate = How many to retain our population for the best ones
        X = our data to test accuracy
        y = our data-label to test accuracy
```

```
        '''

        self.n_pops = n_pops
        self.net_size = net_size
        self.nets = [Network(self.net_size) for i in range(self.n_pops)]
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.retain_rate = retain_rate
        self.X = X[:]
        self.y = y[:]

    def get_random_point(self, type):

        '''
        @type = either 'weight' or 'bias'
        @returns tuple (layer_index, point_index)
            note: if type is set to 'weight', point_index will return (row_index, col_index)
        '''

        nn = self.nets[0]
        layer_index, point_index = random.randint(0, nn.num_layers-2), 0
        if type == 'weight':
            row = random.randint(0,nn.weights[layer_index].shape[0]-1)
            col = random.randint(0,nn.weights[layer_index].shape[1]-1)
            point_index = (row, col)
        elif type == 'bias':
            point_index = random.randint(0,nn.biases[layer_index].size-1)
        return (layer_index, point_index)

    def get_all_scores(self):
        return [net.score(self.X, self.y) for net in self.nets]

    def get_all_accuracy(self):
        return [net.accuracy(self.X, self.y) for net in self.nets]

    def crossover(self, father, mother):

        '''
        @father = neural-net object representing father
        @mother = neural-net object representing mother
        @returns = new child based on father/mother genetic information
        '''

        # make a copy of father 'genetic' weights & biases information
        nn = copy.deepcopy(father)

        # cross-over bias
        for _ in range(self.nets[0].bias_nitem):
            # get some random points
            layer, point = self.get_random_point('bias')
            # replace genetic (bias) with mother's value
```

```
            if random.uniform(0,1) < self.crossover_rate:
                nn.biases[layer][point] = mother.biases[layer][point]

        # cross-over weight
        for _ in range(self.nets[0].weight_nitem):
            # get some random points
            layer, point = self.get_random_point('weight')
            # replace genetic (weight) with mother's value
            if random.uniform(0,1) < self.crossover_rate:
                nn.weights[layer][point] = mother.weights[layer][point]

        return nn

    def mutation(self, child):

        '''
        @child_index = neural-net object to mutate its internal weights & biases value
        @returns = new mutated neural-net
        '''

        nn = copy.deepcopy(child)

        # mutate bias
        for _ in range(self.nets[0].bias_nitem):
            # get some random points
            layer, point = self.get_random_point('bias')
            # add some random value between -0.5 and 0.5
            if random.uniform(0,1) < self.mutation_rate:
                nn.biases[layer][point] += random.uniform(-0.5, 0.5)

        # mutate weight
        for _ in range(self.nets[0].weight_nitem):
            # get some random points
            layer, point = self.get_random_point('weight')
            # add some random value between -0.5 and 0.5
            if random.uniform(0,1) < self.mutation_rate:
                nn.weights[layer][point[0], point[1]] += random.uniform(-0.5, 0.5)

        return nn

    def evolve(self):

        # calculate score for each population of neural-net
        score_list = list(zip(self.nets, self.get_all_scores()))

        # sort the network using its score
        score_list.sort(key=lambda x: x[1])

        # exclude score as it is not needed anymore
        score_list = [obj[0] for obj in score_list]
```

```
        # keep only the best one
        retain_num = int(self.n_pops*self.retain_rate)
        score_list_top = score_list[:retain_num]

        # return some non-best ones
        retain_non_best = int((self.n_pops-retain_num) * self.retain_rate)
        for _ in range(random.randint(0, retain_non_best)):
            score_list_top.append(random.choice(score_list[retain_num:]))

        # breed new childs if current population number less than what we want
        while len(score_list_top) < self.n_pops:

            father = random.choice(score_list_top)
            mother = random.choice(score_list_top)

            if father != mother:
                new_child = self.crossover(father, mother)
                new_child = self.mutation(new_child)
                score_list_top.append(new_child)

        # copy our new population to current object
        self.nets = score_list_top

def main():

    # load data from iris.csv into X and y
    df = pd.read_csv("iris.csv")
    X = df.iloc[:, :-1].values
    y = df.iloc[:, -1].values

    # convert y into one-hot encoded format
    y = y.reshape(-1, 1)
    enc = OneHotEncoder()
    enc.fit(y)
    y = enc.transform(y).toarray()

    # parameters
    N_POPS = 30
    NET_SIZE = [4,6,5,3]
    MUTATION_RATE = 0.2
    CROSSOVER_RATE = 0.4
    RETAIN_RATE = 0.4

    # start our neural-net & optimize it using genetic algorithm
    nnga = NNGeneticAlgo(N_POPS, NET_SIZE, MUTATION_RATE, CROSSOVER_RATE,
RETAIN_RATE, X, y)

    start_time = time.time()

    # run for n iterations
    for i in range(1000):
```

```
        if i % 10 == 0:
            print("Current iteration : {}".format(i+1))
            print("Time taken by far : %.1f seconds" % (time.time() - start_time))
            print("Current top member's network accuracy: %.2f%%\n" % nnga.get_all_accuracy()[0])

        # evolve the population
        nnga.evolve()

if __name__ == "__main__":
    main()
```

Output:

….
…..

```
Current iteration : 961
Time taken by far : 134.7 seconds
Current top member's network accuracy: 98.67%

Current iteration : 971
Time taken by far : 136.2 seconds
Current top member's network accuracy: 98.67%

Current iteration : 981
Time taken by far : 137.7 seconds
Current top member's network accuracy: 98.67%

Current iteration : 991
Time taken by far : 139.2 seconds
Current top member's network accuracy: 98.67%
```