AI – Lab 5_b

Zinat abo hgool 206721714

Elie Haddad 207931536

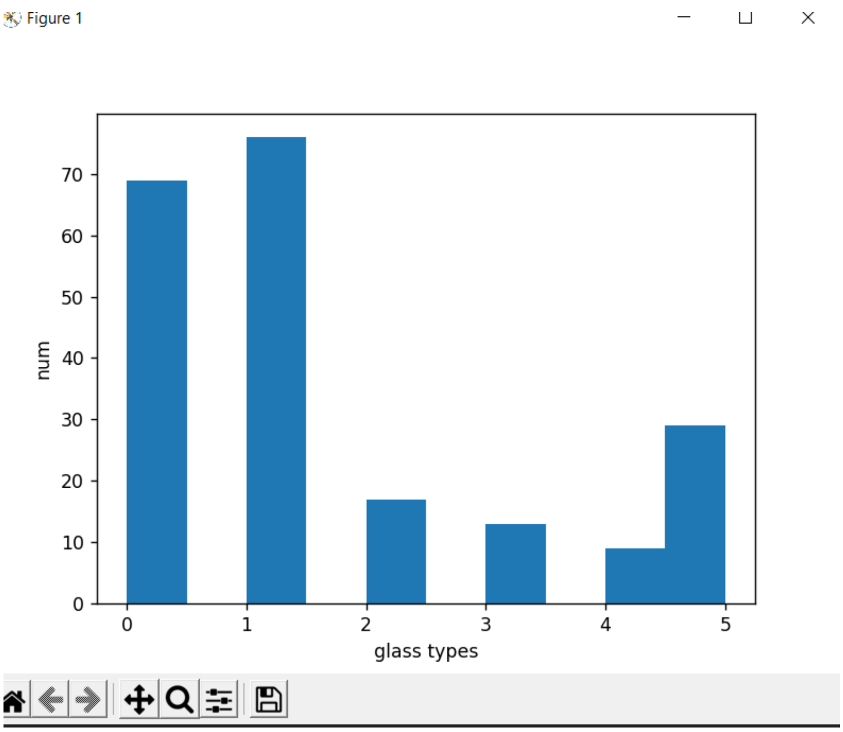
Building the neural network:

First, we downloaded the file "glass.data" form the link in the homework. Then we read the data in the file "readData.py" and we divide the glass types in to 0-5 groups:

```python
class readData:
    def __init__(self):
        self.path='glass.data'

    def readData(self):

        df = pd.read_csv(self.path)
        last_column,data = df.values[:, -1],df.values[:,1:-1]

        print(df)

        labels = LabelEncoder().fit_transform(last_column)
        print(labels)

        y =np.array(labels)
        plt.hist(y)
        plt.xlabel('glass types')
        plt.ylabel('num')
        plt.show()

        # X=df.drop(data)
        X=preprocessing.normalize(data)
```

```python
        plt.ylabel('num')
        plt.show()

        # X=df.drop(data)
        X=preprocessing.normalize(data)
        normalized = DataFrame(MinMaxScaler().fit_transform(data))

        # X=normalize(X)



        # ros = RandomOverSampler(random_state=42)
        # x_ros, y_ros = ros.fit_resample(X, y)

        train_x, test_x, train_y, test_y = train_test_split(normalized,labels,stratify=labels,test_size=0.2,random_state=42)
        print('train_x :',train_x.shape)
        print('train_y :',train_y.shape)
        print('test_x :',test_x.shape)
        print('test_y :',test_y.shape)
        return train_x, train_y, test_x, test_y
```

The graph showing the distribution:



Divide the data into 0.8 training set and 0.2 testing set and

Normalize the data:

```
    plt.ylabel('num')
    plt.show()

    # X=df.drop(data)
    X=preprocessing.normalize(data)
    normalized = DataFrame(MinMaxScaler().fit_transform(data))

    # X=normalize(X)



    # ros = RandomOverSampler(random_state=42)
    # x_ros, y_ros = ros.fit_resample(X, y)

    train_x, test_x, train_y, test_y = train_test_split(normalized,labels,stratify=labels,test_size=0.2,random_state=42)
    print('train_x :',train_x.shape)
    print('train_y :',train_y.shape)
    print('test_x :',test_x.shape)
    print('test_y :',test_y.shape)
    return train_x, train_y, test_x, test_y
```

The neural network:

Our network gets 9 nodes as an input and six logits

We build a class representing the neural network with the following attributes and functions:

We calculated the accuracy of the network,

Here is the implementation of the macro and micro according to the F1 criteria:

```python
import numpy as np
from sklearn.neural_network import MLPClassifier
class NW:
    def __init__(self,train,train_y,test,test_y):
        self.train=train
        self.test=test
        self.train_y=train_y
        self.test_y=test_y
    def mlpFunc(self):
        results = [0, 0, 0, 0, 0, 0]
        correct = [0, 0, 0, 0, 0, 0]
        mlp = MLPClassifier(random_state=1, max_iter=30000)
        mlp= mlp.fit(self.train, self.train_y)
        predict_x = mlp.predict_proba(self.test)
        predict_y=self.softmax(predict_x)
        x = 0
        for p, t in zip(predict_y, self.test_y):
            results[p] += 1
            if p == t:
                correct[p] += 1
                x += 1
        print(f'Micro: {x / 43}')
        print(results)
        counter = 0
        for a, r in zip(results, correct):
            counter += (r / a)
        print(f'Macro: {counter / 6}')
```

Then we implemented the soft max function:

```python
def softmax(self,predict_x):
    predict_y = []
    counter=0
    for i in predict_x:
        index = 0
        max = -1
        marr = np.exp(i) / np.sum(np.exp(i), axis=0)
        m = 0
        print(f'{counter}:    {marr}')
        for j in marr:
            temp=0
            if j > max:
                max=j
                m=index
            index += 1
        predict_y.append(m)
        counter += 1
    print(f'softmax:    {predict_y}')
    return predict_y
```

The accuracy of the network:

```
softmax:   [5, 3, 1, 3, 0, 1, 1, 1, 1, 0, 2, 0, 5, 4, 0, 5, 5, 0, 2, 3, 1, 0, 5, 1, 1, 1, 1, 4, 1, 0, 1, 2, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 4]
Micro: 0.7674418604651163
[11, 18, 3, 3, 3, 5]
Macro: 0.8030303030303031
```

This is the result we got in the micro and macro accuracy.

The genetic algorithm

First we initialized a population and each gene in the population has neural network

```
class Genetic5:
    def __init__(self, args):
        self.best = None
        self.args = args
        self.esize = int(self.args.GA_POPSIZE * self.args.GA_ELITRATE)
        self.population = []
        self.parasites = []
        self.buffer = []

    def init_population(self):  # create popsize citizens

        for i in range(self.args.GA_POPSIZE):  # initialize the agents population
            array = []
            depth = random.randint(1, 10)
            for i in range(depth):
                array.append(random.randint(2, 200))
            if random.randint(0, 2) == 0:
                activate = 'relu'
            else:
                activate = 'tanh'

            agent = Agent(array, 0, NWAgent(depth, array, activate))
            self.population.append(agent)
        # ar=self.population[0].arr
        # f = 100
```

And then we calculated the fitness to be 1 – the accuracy

```
def calc_fitness(self, population: list[Agent]):
    for i in range(self.args.GA_POPSIZE):
        mlp = MLPClassifier(hidden_layer_sizes=self.population[i].network.hidden, max_iter=30000,activation=self.population[i].network.activate, solver='ada
        mlp.fit(self.args.train_x, self.args.train_y)
        conf = confusion_matrix(mlp.predict(self.args.test_x), self.args.test_y)
        sum = conf.sum()
        x = conf.trace()
        self.population[i].fitness = 1 - (x / sum)
        self.population[i].reg = 0
```

Mate:

We implemented mate using parent selection from the top half genes. We used elitism rate of 0.1 and mutating rate of 0.1.

```
def mate(self):
    self.elitism(self.population, self.buffer)
    for i in range(self.esize, self.args.GA_POPSIZE):
        i1 = randint(0, (self.args.GA_POPSIZE / 2) - 1)
        i2 = randint(0, (self.args.GA_POPSIZE / 2) - 1)
        minsize = min(self.population[i1].network.depth, self.population[i2].network.depth)
        # i1 = self.RWS(population, buffer)[0]
        # i2 = self.RWS(population, buffer)[0]
        # i1 = self.tournamentSelection(population)
        # i2 = self.tournamentSelection(population)

        pos = random.randint(0, minsize)
        self.population[i].network.hidden = self.population[i1].network.hidden[0: pos] +self.population[i2].network.hidden[pos:]
        size=len(self.population[i].network.hidden)
        self.population[i].network.depth = size
```

The implementation of the regression function:

```
def regression(self):

    for pop in self.population:
        mlp = MLPClassifier(hidden_layer_sizes=self.population[i].network.hidden, max_iter=30000,
                            activation=self.population[i].network.activate, solver='adam', random_state=1)
        coef = mlp.coefs_
        sum = 0
        mul = 1
        for i in range(len(coef)):
            for j in range(len(coef[i])):
                sum+=coef[i][j] ** 2
        for i in range(pop.network.depth):
              mul*=pop.network.hidden[i]

    return sum*0.5/(2*len(self.args.train_x))*(1/mul)
```

The results we got:

```
       1  1.52101  13.64  4.49  1.10  71.78  0.06  8.75  0.00  0.00.1  1.1
0      2  1.51761  13.89  3.60  1.36  72.73  0.48  7.83  0.00    0.00    1
1      3  1.51618  13.53  3.55  1.54  72.99  0.39  7.78  0.00    0.00    1
2      4  1.51766  13.21  3.69  1.29  72.61  0.57  8.22  0.00    0.00    1
3      5  1.51742  13.27  3.62  1.24  73.08  0.55  8.07  0.00    0.00    1
4      6  1.51596  12.79  3.61  1.62  72.97  0.64  8.07  0.00    0.26    1
..   ...      ...    ...   ...   ...    ...   ...   ...   ...     ...  ...
208  210  1.51623  14.14  0.00  2.88  72.61  0.08  9.18  1.06    0.00    7
209  211  1.51685  14.92  0.00  1.99  73.06  0.00  8.40  1.59    0.00    7
210  212  1.52065  14.36  0.00  2.02  73.42  0.00  8.44  1.64    0.00    7
211  213  1.51651  14.38  0.00  1.94  73.61  0.00  8.48  1.57    0.00    7
212  214  1.51711  14.23  0.00  2.08  73.36  0.00  8.62  1.67    0.00    7

[213 rows x 11 columns]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5]
train_x : (170, 9)
train_y : (170,)
test_x : (43, 9)
test_y : (43,)
0:   [0.12969762 0.12969768 0.12969762 0.13014241 0.12986331 0.35090136]
1:   [0.12991959 0.13120892 0.12991959 0.34877142 0.13023534 0.12994513]
2:   [0.15002955 0.31522252 0.13547816 0.13308664 0.13308803 0.1330951 ]
```

```
0:   [0.12969762 0.12969768 0.12969762 0.13014241 0.12986331 0.35090136]
1:   [0.12991959 0.13120892 0.12991959 0.34877142 0.13023534 0.12994513]
2:   [0.15002955 0.31522252 0.13547816 0.13308664 0.13308803 0.1330951 ]
3:   [0.12957712 0.12957712 0.12957712 0.3520487  0.12957712 0.1296428 ]
4:   [0.30320973 0.16042556 0.13441202 0.13398428 0.13398419 0.13398423]
5:   [0.16898279 0.28936181 0.1367098  0.13497997 0.13498043 0.13498521]
6:   [0.12990953 0.35012397 0.13062201 0.12977922 0.12977905 0.12978621]
7:   [0.13121273 0.33593768 0.13124809 0.13914985 0.13123048 0.13122117]
8:   [0.21095855 0.24117658 0.13712171 0.13691472 0.13691422 0.13691422]
9:   [0.26074183 0.18345133 0.14266661 0.1368414  0.13784973 0.13844909]
10:   [0.17364663 0.14025855 0.27895178 0.13571193 0.13571846 0.13571265]
11:   [0.33345236 0.14057391 0.13166428 0.13143471 0.13143495 0.1314398 ]
12:   [0.1295924  0.1295924  0.1295924  0.12959303 0.12972624 0.35190353]
13:   [0.12970509 0.1297419  0.1297287  0.1297057  0.35084017 0.13027845]
14:   [0.29190849 0.16431348 0.13910157 0.13489242 0.13489202 0.13489202]
15:   [0.13005354 0.13005481 0.13005354 0.13231371 0.13005898 0.34746541]
16:   [0.12956376 0.12956893 0.12956376 0.1295642  0.12956376 0.35217559]
17:   [0.30704775 0.13516319 0.15628127 0.13373126 0.13393517 0.13384137]
18:   [0.17141219 0.13503581 0.28865614 0.13496066 0.13497426 0.13496094]
19:   [0.13383069 0.15937119 0.13383069 0.3051629  0.13397379 0.13383075]
20:   [0.13299107 0.34467259 0.13130481 0.13034105 0.13034491 0.13034556]
21:   [0.32236918 0.14220357 0.13792538 0.1324986  0.13250201 0.13250125]
22:   [0.12956258 0.12956286 0.12956258 0.12956259 0.12956258 0.35218681]
23:   [0.15745284 0.30653837 0.13475463 0.13375087 0.13375091 0.13375237]
24:   [0.21545742 0.23640788 0.13717967 0.13698499 0.13698499 0.13698506]
25:   [0.18242886 0.21383548 0.18690769 0.13894268 0.13894263 0.13894266]
26:   [0.12981902 0.34973622 0.12981902 0.1309877  0.12981902 0.12981902]
27:   [0.12974677 0.1305744  0.12974677 0.12975461 0.35043014 0.12974732]
28:   [0.15163479 0.31284206 0.13355459 0.13345818 0.13330663 0.13520375]
29:   [0.29348746 0.15888395 0.14273189 0.13487507 0.13489543 0.1351262 ]
```

```
main (2)
36:   [0.13245374 0.32560618 0.13384245 0.1321805  0.13218929 0.14372784]
37:   [0.29634557 0.15002957 0.14765519 0.13474678 0.13590244 0.13532044]
38:   [0.13579143 0.27707322 0.18025058 0.13562819 0.13562833 0.13562824]
39:   [0.27349265 0.15858159 0.15887461 0.13635    0.13635017 0.13635098]
40:   [0.33306878 0.13620462 0.13623946 0.13149573 0.13149569 0.13149573]
41:   [0.12992336 0.34872943 0.12992336 0.13157712 0.12992336 0.12992336]
42:   [0.12969556 0.12985062 0.1296954  0.129695   0.35092646 0.13013696]
softmax:   [5, 3, 1, 3, 0, 1, 1, 1, 1, 0, 2, 0, 5, 4, 0, 5, 5, 0, 2, 3, 1, 0, 5, 1, 1, 1, 1, 4, 1, 0, 1, 2, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 4]
Micro: 0.7674418604651163
[11, 18, 3, 3, 3, 5]
Macro: 0.8030303030303031
fitness:  0.2093023255813954
best agent:  [114, 187, 170, 17, 152, 31, 32, 122, 24, 152]
Clock ticks time:  37.2255539894104
fitness:  0.2093023255813954
best agent:  [114, 187, 170, 17, 152, 31, 32, 122, 24, 152]
Clock ticks time:  35.06525707244873
fitness:  0.2093023255813954
best agent:  [114, 187, 170, 17, 152, 31, 32, 122, 24, 152]
Clock ticks time:  37.48623275756836
fitness:  0.2093023255813954
best agent:  [114, 187, 170, 17, 152, 31, 32, 122, 24, 152]
Clock ticks time:  36.32174754142761
fitness:  0.2093023255813954
best agent:  [114, 187, 170, 17, 152, 31, 32, 122, 24, 152]
Clock ticks time:  45.188724517822266
fitness:  0.2093023255813954
best agent:  [114, 187, 170, 17, 152, 31, 32, 122, 24, 152]
```

This is a little bit of our main:

Using this function we run the genetic algorithm

```
def run(ga, args1, esize, start_elapsed):
    lst = []


    ga.calc_fitness(ga.population)
    ga.sort_by_fitness(ga.population)
    bestfitness=ga.population[0].fitness
    ga.best = ga.population[0]
    for i in range(args1.maxiter):
        iteration_time = time.time()
        ga.calc_fitness(ga.population)
        ga.sort_by_fitness(ga.population)
        if ga.population[0].fitness < bestfitness:
            ga.best=ga.population[0]
            ga.best=Agent(ga.population[0].arr,ga.population[0].fitness,ga.population[0].network)
            ga.best.age=ga.population[0].age
            ga.best.reg=ga.population[0].reg
            bestfitness=ga.population[0].fitness
        # ga.print_best(ga.population)
        print("fitness: ", bestfitness)
        print("best agent: ",ga.best.arr)

        lst.append(bestfitness)
```

And here we called the function that read and divide the data into training and test set and then we sent the result to the genetic algorithm:

```python
def get_input():
    popsize = 10
    maxIter = 100
    rd = readData()

    train_x, train_y, test_x, test_y = rd.readData()
    nw = NW(train_x, train_y, test_x, test_y)
    nw.mlpFunc()

    args = Var(maxIter, popsize, 0.1, 0.1,train_x, train_y, test_x, test_y)
    ARGS = args
    return args
```

<u>Improvements in calculating the fitness:</u>

A) parallelization:

Parallelization can be implemented using threads, and constructing a thread pool. This may create a significant memory management overhead, due to allocating and deallocating many thread objects, but it helps us in running more than one neural network. It also helps in running more iterations in our algorithm. So, in general, threads and using parallelization can be helpful in improving the overhead od calculating the fitness.

B) cached results:

Cache partial results from the gene fitness computation that was already been calculated, can reduce future fitness computations time.

c)reduce the using of floating point and use instead complete numbers:

if our algorithm doesn't need to be precise then we can use this way to improve the execution time of the whole algorithm, if not we prefer to use another way.

d)smart initialization:

if we can initialize the population in specific way, we can increase the quality of the classification. This can be done in initializing some neural networks and then use them to generate initial population.