Université de Limoges | FACULTÉ DES SCIENCES ET TECHNIQUES

M1 CRYPTIS
Artificial Intelligence 2 Project

Authors

TRAN Ngoc Nhat Huyen - 20196635
NGUYEN Phuong Hoa - 20196637

May 15, 2020

Contents

List of Figures

List of Tables

1 Description

This project aims to implement Supervised and unsupervised learning methods. Start by generating four groups of 50 points each in 2D space whose coordinates are between 0 and 1 and randomly generated.

Section 3 we present K-means which is the simplest Unsupervised learning algorithms and the way we implemented them. Section 4 focuses on Supervised learning, specially for *Linearly separable classification* using a perceptron and *Non-linearly separable classification* using a neural network.

All algorithms is implemented in Python Language.

2 Generate data

We randomly generated n points, each in 2D space whose coordinates are between 0-1. Apply a different translation to each group so as to obtain the points distributed as follows:

```
def gen_data(n):
    X = []
    group1 = np.random.uniform(low=0, high=1, size=(n, 2))
    for x, y in group1: X.append([0, 1 + x, 1 + y])
    for x, y in group1: X.append([0, 1 + x, -2 + y])
    for x, y in group1: X.append([0, -2 + x, 1 + y])
    for x, y in group1: X.append([0, -2 + x, -2 + y])
    data = np.array(X)
    clusters = data[:, 0]
    data = data[:, 1:]
    return data, clusters
```

3 Unsupervised learning by using K-means

3.1 The data structures used

As proposed in K-means method, we define some variable as follows:

- $data[i][]$ represents the coordinate of the point i by $(data[i][0], data[i][1])$
- $clusters[i]$ is which cluster the point i was assigned.
- $nb\_cluster$ is an array contains the number of elements in each cluster
- k is number of groups
- I_{b_max} is max error of inter-class inertia

3.2 Reminder of the learning methods used

K-means problem is to find cluster centers that maximize the inter-class variance. So, we define `inter_class` function to calculate inter-class variance as follows:

```
1  def inter_class(nb, centroids, g):
2      I_b = 0
3      for i in range(len(nb)):
4          d = dist(centroids[i], g)
5          I_b += nb[i] * d * d
6      return I_b
```

In this function, *nb* is an array contains the number of elements in each cluster. *centroids* are current centroids of the each cluster. g is center of gravity. dist(A,B) is distance from point A to point B. We calculated according to the following formula: $I_B = \sum_{i=1}^{k} \times |C_i| \times d^2(g_i, g)$

Here is pseudo code which runs K-means on a dataset. It is a short algorithm made longer by verbose commenting.

```
1  while (iteration > 0): # run on several times to reach greatest inter_class
2      # inittial random center
3      centroids = data[np.random.choice(data.shape[0], k, replace=False)]
4      while True:
5          # assign observation
6          for i in range(len(data)):
7              dist_centroids = [dist(data[i], x) for x in centroids]
8              cluster = np.argmin(dist_centroids)
9              clusters[i] = cluster
10         old_ib = I_b
11         # recalculate new centroids
12         for i in range(k):
13             groups = [data[j] for j in range(len(data)) if clusters[j] == i]
14             nb_cluster[i] = len(groups)
15             centroids[i] = np.mean(groups, axis=0)
16         # calculate the mean vector G of the centroids
17         g = np.mean(centroids, axis=0)
18         I_b = inter_class(nb_cluster, centroids, g)
19         if (I_b <= old_ib): break
20     if (I_b > I_b_max):
21         I_b_max = I_b
22         res_clusters = deepcopy(clusters)
23     iteration -= 1
```

Lines 6-9, for each element in the dataset, we chose the closest centroid, then make that centroid the element's label. Lines 12-15, centroid will be recalculated. Each centroid is the geometric mean of the points that have that centroid's label. For each value of k, run the algorithm on several initialization cases then keep the result corresponding to a value of the greatest `inter_class` inertia. In this case, we set *iteration* to 5.

3.3 The results obtained

Figure 1 displays a generated data. Figure 2 - 4 shows the results when tested k = 2,3,4. We use Gnuplot to display the proposed classification by the algorithm and visually compare with the "natural" classification. Please see the detailed demo in the following link: `https://youtu.be/p7X1dzME4h4`

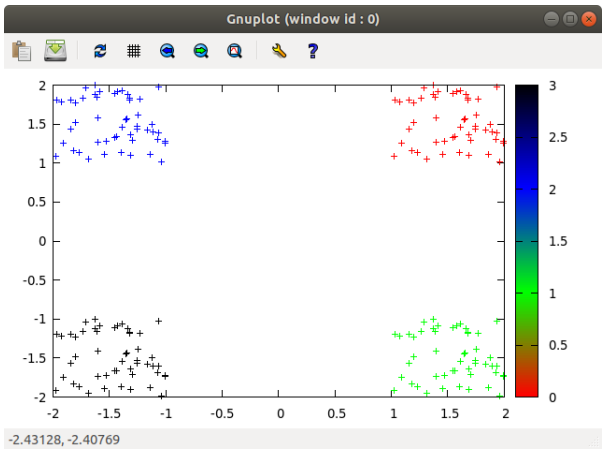

Figure 1: Generated data

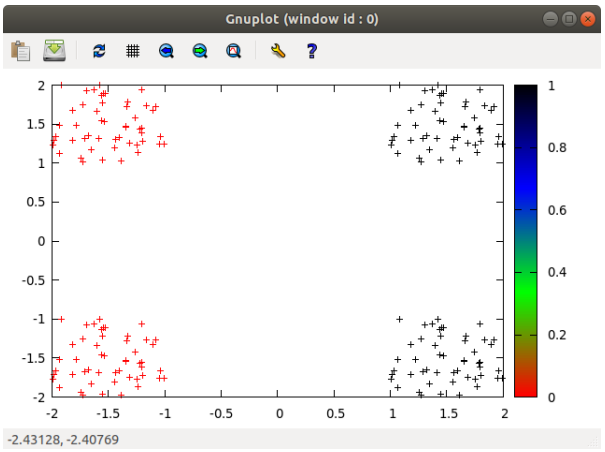

Figure 2: KMeans algorithm with k=2

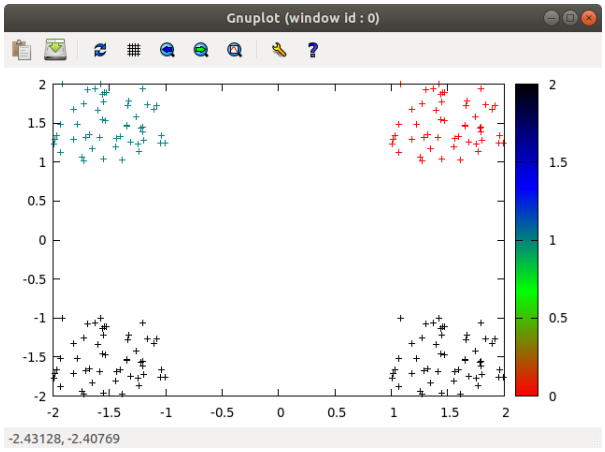

Figure 3: KMeans algorithm with k=3

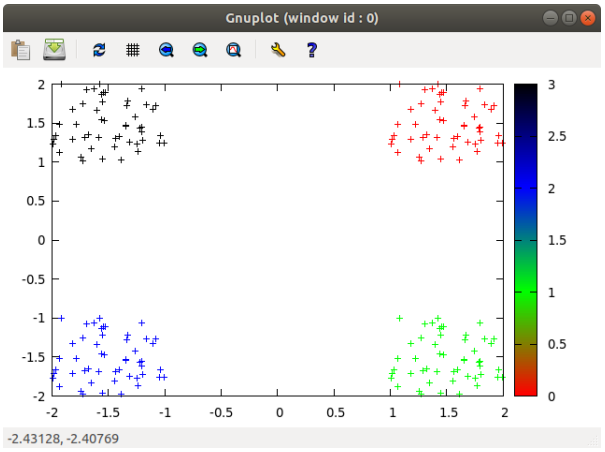

Figure 4: KMeans algorithm with k=4

4 Supervised learning

4.1 Perceptron Learning Algorithm

4.1.1 Reminder of basic knowledge

The perceptron is an algorithm for learning a binary classifier called a threshold function: a function that maps its input **x** (a real-valued vector) to an output value.

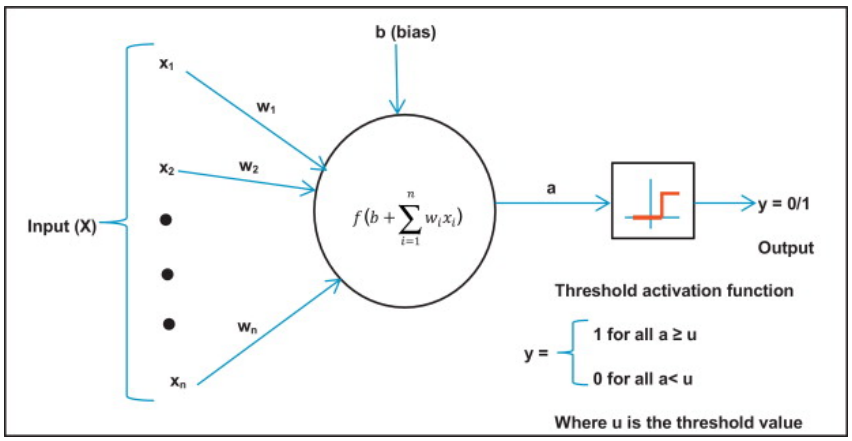

Figure 5: The perceptron model

Threshold function is defined:

$$h_{\mathbf{w}}(\mathbf{x}) = Threshold(\mathbf{w}.\mathbf{x}) \text{ where } Threshold(z) = 1 \text{ if } z \geq 0 \text{ and } 0 \text{ otherwise.}$$

In this algorithm, there is a simple weight update rule that converges to a solution - a linear separator that classifies the data. This rule is called the **perceptron learning rule**:

$$w_i \leftarrow w_i + \alpha(y - h_{\mathbf{w}}(\mathbf{x})) \times x_i$$

The parameter α is call **learning rate**.

4.1.2 Implementation

```
def threshold(w, x, b):
    t = w[0] * x[0] + w[1] * x[1] + b
    if t >= 0: return 1
    else:      return 0
def perceptron_learning(data, class_label):
    # Initialize parameter randomly
    w = [random.uniform(-1, 1), random.uniform(-1, 1)]
    b = random.uniform(-1, 1)  #bias
    alpha = 0.1 #learning rate
    for each (x,y) in (data, class_label):
        #calculate predict label
        h = threshold(w, x, b)
        if (h !=y):
            #update rule
            w = w + alpha.(y-h).x
            b = b + alpha.(y-h)
    return w, b
```

4.2 Logistic Function

$$Logistic(z) = \frac{1}{1+e^{-z}}$$

With the logistic function replacing the hard-threshold function, we now have:

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w}.\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}.\mathbf{x}}}$$

4.2.1 Implementation

Firstly, the program need to execute the logistic learning function on training data set to compute parameter w and b.

```
1  def logistic_learning(training_data, training_class):
2      # w is random initial vector
3      w = [random.uniform(-1, 1), random.uniform(-1, 1)]
4      alpha = 0.1  # learning rate
5      b = random.uniform(-1, 1)
6      foreach (x,y) in (training_data, training_class):
7          # calculate h_w(x_i)
8          z = w[0] * x[0] + w[1] * x[1] +b
9          hw, predictY = logistic(z)
10         # if predict class is different from original class
11         if predictY != y:
12             #update weight and bias
13             w_0 = w[0] + alpha * (y - hw) * x[0]
14             w_1 = w[1] + alpha * (y - hw) * x[1]
15             w = [w_0, w_1]
16             b = b + alpha * (y - hw)
17     return w, b
```

As below, the logistic function calculates predict class label $predictY$ and h_w. At line 2, we use Python built-in library *math* to obtain e^{-z}.

```
1  def logistic(z):
2      h = 1 / (1 + math.exp(-z))
3      if h >= 0.5:
4          return h, 1
5      else:
6          return h, 0
```

Finally, we apply results of w and b to calculate *error rate* on test data set.

```
1  def test_logistic(test_data, test_class, w, b):
2      misclassified = 0
3      foreach (x,y) in (test_data, test_class):
```

```
        z = w[0] * x[0] + w[1] * x[1]+b
        h, label = logistic(z)
        if label != y:
            misclassified += 1
    print("Logistic function misclassified", misclassified)
    return misclassified / len(test_data)
```

4.3 Neural network

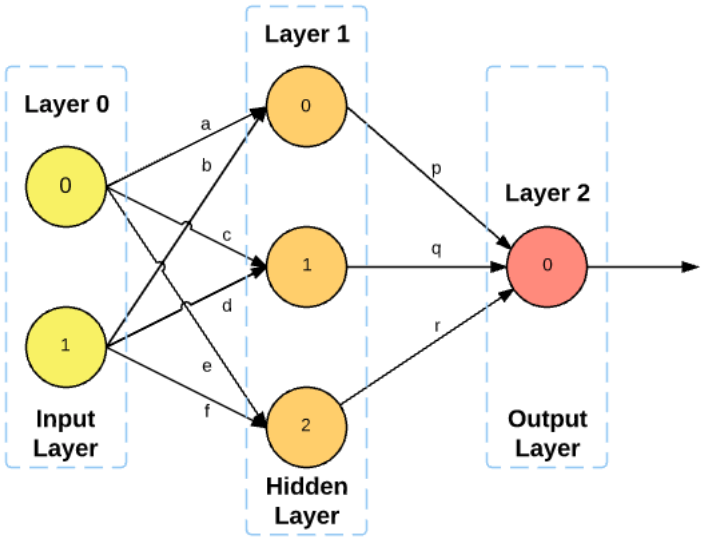

Figure 6: Neural network structure

4.3.1 Reminder of basic knowledge

Neural networks are composed of artificial neurons or nodes interconnected by directed links. Each node in layers can be called an *unit* or a *neuron*. Output of an unit is doted a (represents *activation*), which means that the unit's value is after applying *activation* function. Weight $w_{i,j}$ determined the strength of each link that from unit i to unit j. In addition, we also use bias b.

Activation functions are mathematical equations that define the output of a node in network based on the formulas:

$$in_j = \sum w_{ij} a_i$$
$$a_j = g(in_j) = g(\sum w_{ij} a_i)$$

In this project, it is *logistic function* as activation function g. So $a_j = Logistic(\sum w_{ij} a_i)$

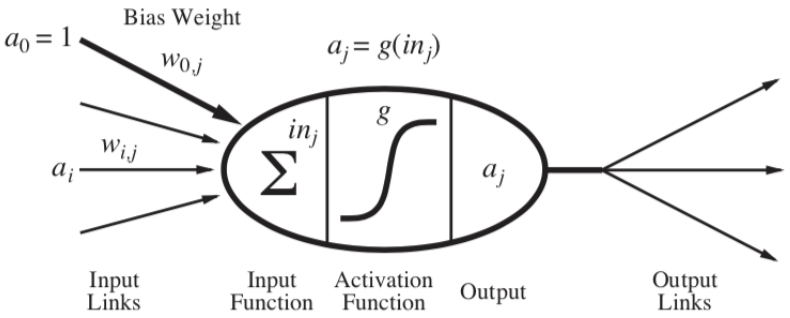

Figure 7: A simple mathematical model for a neuron

4.3.2 Implementation

Training data is composite of input and output samples which are labeled correctly. However, they cannot show what value the hidden nodes should be. We use *Back-propagation* algorithm to compute the error from the output layer to the hidden layers, then apply the weight-update rule.

```
def back_prop_learning(training_data, training_label):
    n = a random value for number of nodes in hidder layer
    w = random initialization for all value w_ij
    b = random initialization for all bias to pass to node j
    alpha = 0.1 #learning rate
    for each (x,y) in (training_data, training_class):
        #PROPAGATE INPUT FORWARD TO OUTPUT
        for each node in 2 nodes of input layer
            a[i] = x[i]
        for each node j in hidden layer and output layer:
            vector_in[j] = vector_in[j] + w[i][j] * a[i]
            a[j] = logistic(vector_in[j] + b[j])
        #PROPAGATE DELTA BACKWARD
        #for node in output layer
        delta[j] = y - a[n+2]
        for each node i in input layer and input layer:
            g = logistic(vector_in[i] +b[i])
            delta[i] = g*(1-g)*sum([w[i][j]*delta[j] for j in range(2,
            ↪  n+2)])
        #update weight
        for each w_ij in network:
            w_ij = w_ij + alpha*a[i]*delta[j]
        #update bias
        for each bias into node j:
            b[j] = b[j] + alpha*delta[j]
return w, b, n
```

4.4 The result obtained

Please check the demonstration video at Youtube link `https://youtu.be/p7X1dzME4h4`

4.4.1 Requirement 1

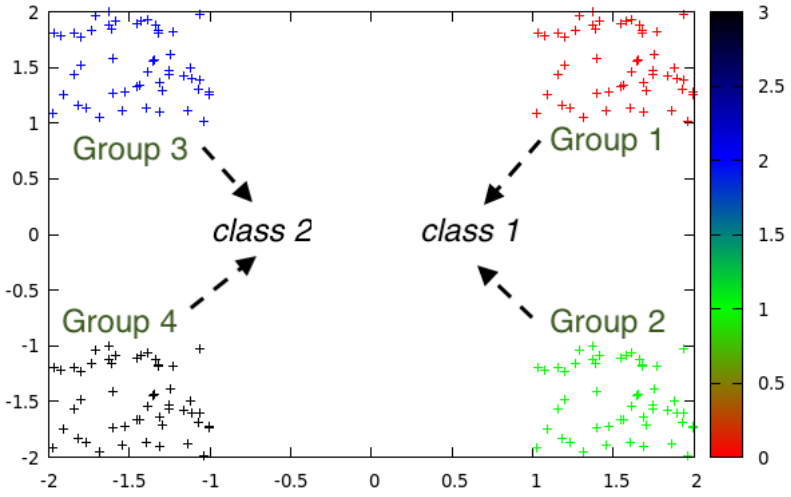

Figure 8: Visualization for Requirement 1

In this requirement, class 1 is composed of the points of group1 and group2, and class 2 is composed of the points of group3 and group4. We proceed testing on 5 sets of training and testing data. They are visualized in Figure 8.

Data set	Perceptron		Logistic	
	Misclassified	Error rate	Misclassified	Error rate
1	0	0	0	0
2	1	0.0066	0	0
3	0	0	0	0
4	0	0	2	0.0132
5	0	0	0	0
6	0	0	0	0

Table 1: Performance of Perceptron and Logistic Function

For this requirement, we will see that both perceptron and logistic's results are similar in this case. The average error rate approaches 0. It is obvious that these classes are easy to separate linearly.

4.4.2 Requirement 2

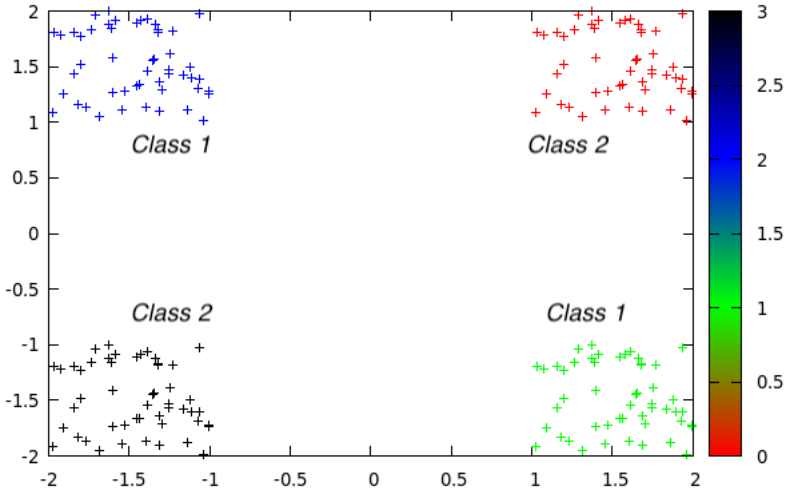

Figure 9: Visualization for Requirement 2

In this requirement, class 1 is composed of the points of group2 and group3, and class 2 is composed of the points of group1 and group4. We proceed testing on 6 sets of training and testing data. In Figure 9, it is impossible to find a linear classifier. In other words, the data cannot be linearly separable. That leads to the difference between two methods. Neural network seems better than perceptron in some cases, and worse in other cases. If there is a larger amount of data, it could be predicted that the advantage belongs to the neural network, especially with more hidden layer. Moreover, performance is also depend on the random initialization value of the variable w and b.

Data set	Perceptron		Neural network	
	Misclassified	Error rate	Misclassified	Error rate
1	71	0.355	88	0.44
2	100	0.5	79	0.395
3	63	0.315	55	0.275
4	67	0.335	74	0.37
5	53	0.265	101	0.505
6	86	0.43	50	0.25

Table 2: Performance of Perceptron and Neural Network