HW5

March 20, 2019

0.0.1 Homework 5

Erin Witmer CSC 440

[10.2] Suppose that the data mining task is to cluster points (with(x,y) representing location) into three clusters, where the points are A1(2,10), A2(2,5), A3(8,4), B1(5,8), B2(7,5), B3(6,4), C1(1,2), C2(4,9). The distance function is Euclidean distance. Suppose initially we assign A1, B1, and C1 as the center of each cluster, respectively. Use the k-means algorithm to show only:

(a) The three cluster centers after the first round of execution.

```
In [1]: names = ['a1','a2','a3','b1','b2','b3','c1','c2']
        x = [2,2,8,5,7,6,1,4]
        y = [10,5,4,8,5,4,2,9]
        centers = [[2,10],[5,8],[1,2]]

In [2]: def dist(obj1, obj2, l):
            total = 0
            for i in range(len(obj1)):
                total += abs(obj1[i]-obj2[i]) ** l
            return total ** (1/l)

In [3]: def assign(centers, x, y, names):
            k = [[],[],[]]

            for i in range(len(names)):
                center = 0;
                min = 100
                for j in range(3):
                    distance = dist((x[i],y[i]),(centers[j][0],centers[j][1]),2)
                    if distance < min:
                        min = distance
                        center = j
                k[center].append(names[i])

            return k
```

After the first execution, the clusters are: [['a1'], ['a3', 'b1', 'b2', 'b3', 'c2'], ['a2', 'c1']]

```
In [4]: cluster_1 = assign(centers,x,y,names)
        print(cluster_1)
```

```
[['a1'], ['a3', 'b1', 'b2', 'b3', 'c2'], ['a2', 'c1']]
```

```
In [5]: def calc_means(clusters, x, y, names):
            centers = [[],[],[]]
            for i in range(len(clusters)):
                x_total = 0
                y_total = 0
                for j in range(len(clusters[i])):
                    x_total += x[names.index(clusters[i][j])]
                    y_total += y[names.index(clusters[i][j])]
                centers[i] = [x_total/len(clusters[i]),y_total/len(clusters[i])]
            return centers
```

```
In [6]: centers_2 = calc_means(cluster_1, x, y, names)
        print(centers_2)
```

```
[[2.0, 10.0], [6.0, 6.0], [1.5, 3.5]]
```

After the first execution, the centers of the clusters are: [[2.0, 10.0], [6.0, 6.0], [1.5, 3.5]]

(b) The final three clusters.

```
In [7]: cluster_2 = assign(centers_2,x,y,names)
        print(cluster_2)
```

```
[['a1', 'c2'], ['a3', 'b1', 'b2', 'b3'], ['a2', 'c1']]
```

```
In [8]: centers_3 = calc_means(cluster_2, x, y, names)
        print(centers_3)
```

```
[[3.0, 9.5], [6.5, 5.25], [1.5, 3.5]]
```

```
In [9]: cluster_3 = assign(centers_3,x,y,names)
        print(cluster_3)
```

```
[['a1', 'b1', 'c2'], ['a3', 'b2', 'b3'], ['a2', 'c1']]
```

```
In [10]: centers_4 = calc_means(cluster_3, x, y, names)
         print(centers_4)
```

```
[[3.6666666666666665, 9.0], [7.0, 4.333333333333333], [1.5, 3.5]]
```

```
In [11]: cluster_4 = assign(centers_4,x,y,names)
         print(cluster_4)
```

```
[['a1', 'b1', 'c2'], ['a3', 'b2', 'b3'], ['a2', 'c1']]
```

These three clusters represent a local optimal solution: [['a1', 'b1', 'c2'], ['a3', 'b2', 'b3'], ['a2', 'c1']]

[10.3] Use an example to show why the k-means algorithm may not find the global optimum, that is, optimizing the within-cluster variation.

```
In [12]: names = ['a1','a2','a3', 'a4']
         x = [0,0,4,4]
         y = [0,2,0,2]
         centers = [[2,0],[2,1]]

In [13]: def assign2(centers, x, y, names):
             k = [[],[]]

             for i in range(len(names)):
                 center = 0;
                 min = 100
                 for j in range(2):
                     distance = dist((x[i],y[i]),(centers[j][0],centers[j][1]),2)
                     if distance < min:
                         min = distance
                         center = j
                 k[center].append(names[i])

             return k

In [14]: cluster_a = assign2(centers, x, y, names)
         print(cluster_a)
```

```
[['a1', 'a3'], ['a2', 'a4']]
```

```
In [15]: def calc_means2(clusters, x, y, names):
             centers = [[],[]]
             for i in range(len(clusters)):
                 x_total = 0
                 y_total = 0
                 for j in range(len(clusters[i])):
                     x_total += x[names.index(clusters[i][j])]
                     y_total += y[names.index(clusters[i][j])]
                 centers[i] = [x_total/len(clusters[i]),y_total/len(clusters[i])]
             return centers
```

```
In [16]: centers_a = calc_means2(cluster_a, x, y, names)
         print(centers_a)

[[2.0, 0.0], [2.0, 2.0]]


In [17]: assign2(centers_a, x, y, names)

Out[17]: [['a1', 'a3'], ['a2', 'a4']]
```

The global optimum is [[’a1’, ’a2’],[’a3’,’a4’]] however because of where the initial centers were placed, the local optimum will be reached but the global optimum will not.

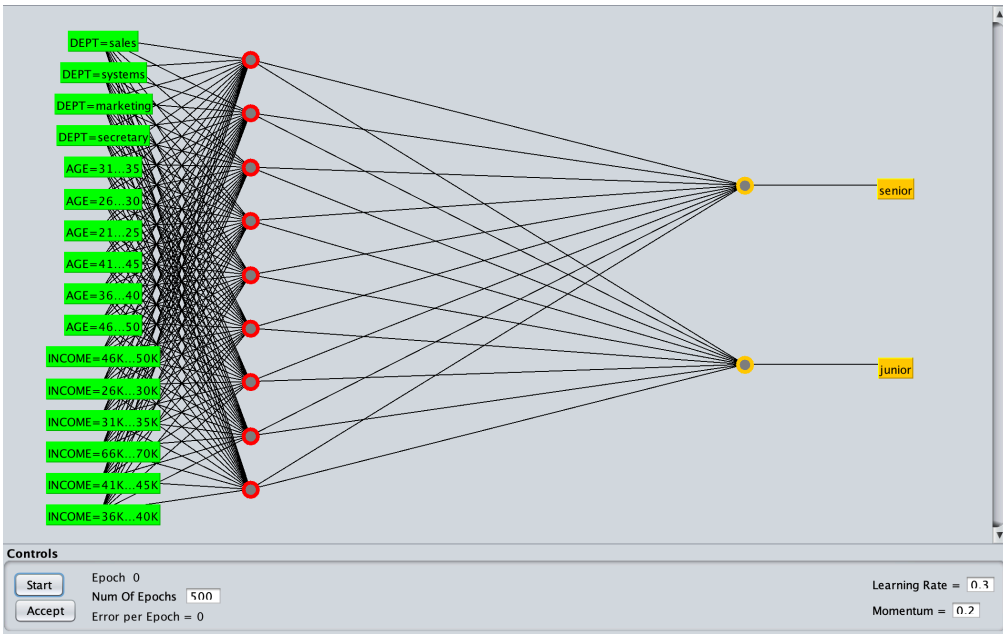
[10.6] Both k-means and k-medoids algorithms can perform effective clustering.

(a) Illustrate the strength and weakness of k-means in comparison with the k-medoids algorithm. These distance based methods use iterative relocation techniques to move objects from one partition to another. Both work well finding spherical clusters in small-mid sized databases. K-means partitions objects into k groups, then: computes the centroid, assigns the data points to the nearest centroid and repeats until an optimal point is found. However, this may be a local optimal point, not global. The main strength of K-means is that it is efficient O(tkn). The main weaknesses are that it often terminates at a local optimal point, it is only applicable to continuous n-dimensional space, you need to specify k, it is sensitive to outliers, not good with non-convex shapes, and is sensitive to the seed point. K-medoids (PAM): is more complex, and therefore less efficient. Instead of taking the mean, it takes a center data point, partitions the data, tests a different non-medoid point, if it reduces the SSE, it reassign the medoid. K-medoids works well for small data sets and is less sensitive to outliers, however it is not efficient for large data sets.

(b) Illustrate the strength and weakness of these schemes in comparison with a hierarchical clustering scheme (e.g., AGNES). Hierarchical methods divides the data into a tree of clusters. Strengths: it does not require K as an input, it is generally efficient for smaller sets, and it can find clusters of arbitrary shapes. The major weaknesses are that the iterations cannot be undone and does not always scale well because each decision of merge or split needs to examine and evaluate many objects or clusters. Variations of these methods such as BIRCH overcome these two weaknesses, but BIRCH only handles numeric data. The partitioning methods described above main strengths vs. this method are ease, efficiency, and the ability to reverse iterations while relative weaknesses are tendency to terminate at a local optimal point, applicability to continuous n-dimensional space, a need to specify k, and difficulty with non-convex shapes.

[9.1] The following table consists of training data from an employee database. The data have been generalized. For example, “31 ... 35” for age represents the age range of 31 to 35. For a given row entry, count represents the number of data tuples having the values for department, status, age, and salary given in that row. Let status be the class-label attribute.

(a) Design a multilayer feed-forward neural network for the given data. Label the nodes in the input and output layers.



Multilayer Forward Feed Network

(b) Using the multilayer feed-forward neural network obtained in (a), show the weight values after one iteration of the back propagation algorithm, given the training instance "(sales,senior,31...35,46K...50K)". Indicate your initial weight values and biases and the learning rate used. Please see attached documents with output of Weka implementations of two different classification models: SVM (results_svm.pdf) and multilayer feed forward neural network (results_mlp.pdf).