Andrew Floyd

November 18th, 2018

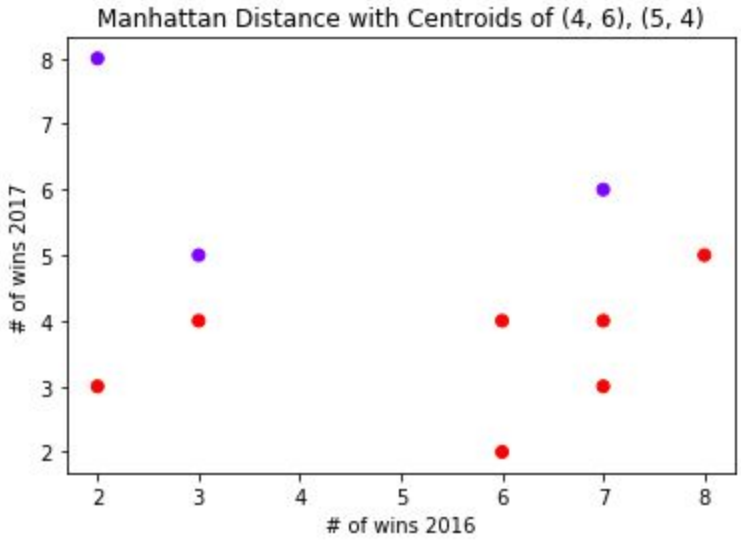
CS3001: Intro to Data Science

Dr. Fu

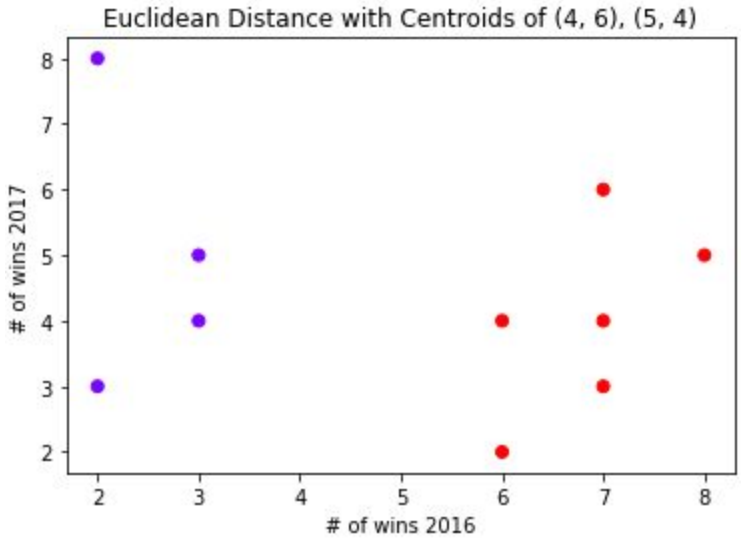
HW7 - KMeans

Github repo: https://github.com/afzm4/cs3001hw7

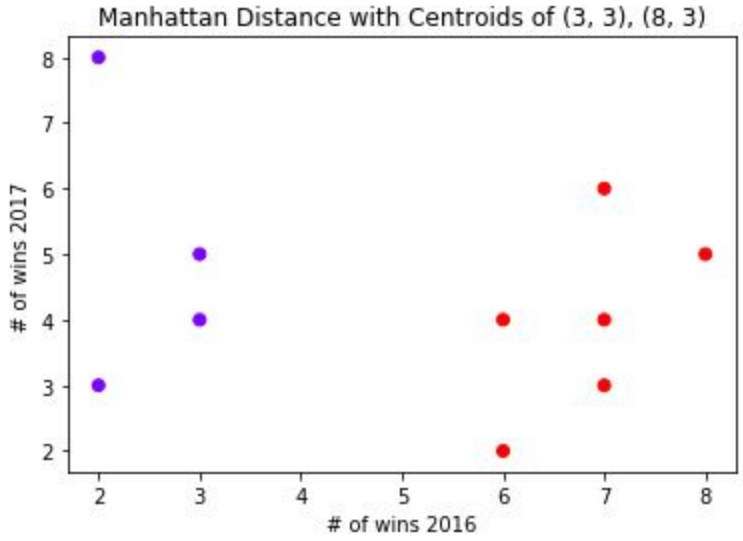
1. This is Manhattan Distance with centers of (4,6) and (5,4).



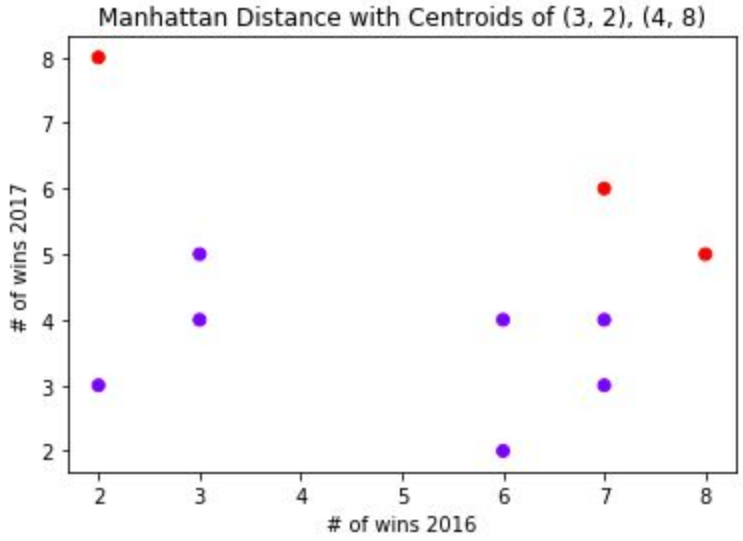
2. Euclidean Distance with centers of (4,6) and (5,4).



3. Manhattan Distance with centers of (3,3) and (8,3).



4. Manhattan Distance with centers of (3,2) and (4,8).



Here a code snippet from these visualizations:

```
clusterer = KMeansClusterer(2, distance=cityblock, initial_means=init)
clusters = clusterer.cluster(football, True, trace=True)
print('Clustered:', football)
print('As:', clusters)
print('Means:', clusterer.means())
print()
plt.figure(0)
plt.title('Manhattan Distance with Centroids of (4, 6), (5, 4)')
plt.xlabel('# of wins 2016')
plt.ylabel('# of wins 2017')
plt.scatter(football[:,0],football[:,1], c=clusters, cmap='rainbow')
```

Some observations seemed to be that Manhattan and Euclidean distance produced different clusters despite using the same centroids. Also, looking at the 3rd and 4th graphs, we see that when we feed it different centroids, different clusters appear despite both of them using Manhattan distance as the their distance metric.

2.

1. By comparing the SSE's of the Euclidean, Cosine and Jaccard K-Means, we see that SSE for the Euclidean is consistently lower than that of Cosine and Jaccard. This would imply that the Euclidean metric is the most effective out of these three. Below is the output of the three including the SSE and centroid values that the code calculated. It appears that Euclidean was the best, followed by Cosine and Jaccard.

   Euclidean:

```
Means: [array([6.29361702, 2.9       , 4.95106383, 1.72978723]), array([5.58      , 2.63333333, 3.98666667, 1.23333333]),
array([7.08695652, 3.12608696, 6.01304348, 2.14347826]), array([5.006, 3.418, 1.464, 0.244])]
SSE:  84.67740539583278
```

   Cosine:

```
Means: [array([5.006, 3.418, 1.464, 0.244]), array([5.94090909, 2.77045455, 4.19772727, 1.29318182]), array([6.35945946, 3.07027027,
5.35945946, 2.06486486]), array([6.81578947, 2.72105263, 5.66315789, 1.80526316])]
SSE:  100.35587341139752
```

   Jaccard:

```
Means: [array([5.84333333, 3.054     , 3.75866667, 1.19866667]), array([6.30625, 3.025  , 4.8375 , 1.70625]), array([5.25384615,
3.00769231, 2.36923077, 0.63846154]), array([6.06911765, 3.1       , 4.01911765, 1.27647059]), array([5.9       , 3.00555556,
4.21111111, 1.42222222]), array([5.4625 , 3.53125, 2.2625 , 0.53125]), array([6.8625, 3.05  , 5.725 , 2.05  ]), array([6.65      , 2.95
, 5.58333333, 1.96666667])]
SSE:  291.4551238555538
```

2. For this question, I calculated the accuracy of each datapoint by taking the difference of each datapoint from it's estimated value, based on the clusters that the program output. I then took the absolute value of the difference as a percentage of the original data value. My results showed me that Euclidean was slightly better than Cosine (as we saw earlier), and that Jaccard came third in accuracy. While this is a different metric of accuracy than SSE, it seems to rank them in a similar fashion. On the Jaccard run, it seems that the pedal width and length seem to have uses with it's clustering, giving it a much higher average error.

Euclidean Accuracy Results:

```
0.05269525382056381
0.07354305244803559
0.08648288368264986
0.21478137459537516
Total Error(%):  0.1068756411366561
```

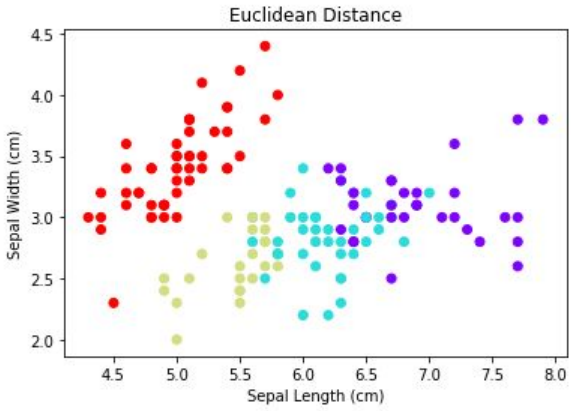
Cosine Accuracy Results:

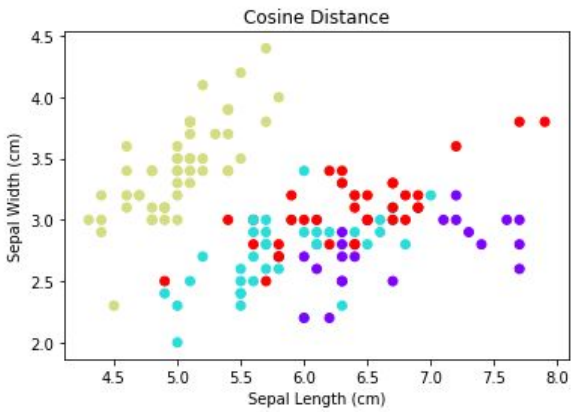
```
0.06803989404251494
0.08437579125334435
0.08602048406129535
0.21270131742075446
Total Error(%):  0.11278437169447728
```

Jaccard Accuracy Results:

```
0.11998033532123839
0.11177911893850374
0.671318072740534
1.7673868143410334
Total Error(%):  0.6676160853353275
```

Here are some charts visualized with 2 of the 4 vector values:





Here the code for this section as well:

```
total1 = 0
total2 = 0
total3 = 0
total4 = 0
for c in range(0,150):
    if clusters[c] == 0:
        ed = data[c]-clusterer.means()[0]
        ed = abs(ed)
        total1 = total1 + (ed[0]/data[c][0])
        total2 = total2 + (ed[1]/data[c][1])
        total3 = total3 + (ed[2]/data[c][2])
        total4 = total4 + (ed[3]/data[c][3])
    elif clusters[c] == 1:
        ed = data[c]-clusterer.means()[1]
        ed = abs(ed)
        total1 = total1 + (ed[0]/data[c][0])
        total2 = total2 + (ed[1]/data[c][1])
        total3 = total3 + (ed[2]/data[c][2])
        total4 = total4 + (ed[3]/data[c][3])
    elif clusters[c] == 2:
        ed = data[c]-clusterer.means()[2]
        ed = abs(ed)
        total1 = total1 + (ed[0]/data[c][0])
        total2 = total2 + (ed[1]/data[c][1])
        total3 = total3 + (ed[2]/data[c][2])
        total4 = total4 + (ed[3]/data[c][3])
    elif clusters[c] == 3:
        ed = data[c]-clusterer.means()[3]
        ed = abs(ed)
        total1 = total1 + (ed[0]/data[c][0])
        total2 = total2 + (ed[1]/data[c][1])
        total3 = total3 + (ed[2]/data[c][2])
        total4 = total4 + (ed[3]/data[c][3])


print(total1/150)
print(total2/150)
print(total3/150)
print(total4/150)
print("Total Error(%): ", (total1/150+total2/150+total3/150+total4/150)/4)
```

3. While this metric does tend to vary each time I run the code, in general it appears that Jaccard requires the most iterations (followed by Cosine and then Euclidean with the least). This seems to be in reverse order of SSE, which would make sense as the more defined the clusters are, the less iterations may be required.

   Below are some screenshots of how many iterations ran:

   Euclidean:

```
iteration
iteration
iteration
iteration
iteration
iteration
```

   Cosine:

```
iteration
iteration
iteration
iteration
iteration
iteration
iteration
iteration
iteration
```

   Jaccard:

```
iteration
iteration
iteration
iteration
iteration
iteration
iteration
iteration
iteration
iteration
iteration
```

4.

- No Change in Centroid Position:

  In this case we see little to no change in SSE for all three methods, but we do see a more different images in the 2-d distribution charts for Cosine especially. The amount of iterations is similar to before as well.

- SSE Value increases in the next iteration:

  In this case, the SSE's of all three are higher than they were when running normally, with this termination condition.

- 100 Iterations are complete:

  In this case, the SSE of the Euclidean wen down slightly (by only a couple of points). I ran each of these a couple times to establish more of an average, since the results did vary from trial to trial (like every other metric). The cosine SSE average was also a bit lower than before, while the Jaccard was all over the place (but eventually about the same as before).

3. Understanding K-Means:

- K-Means clustering is a method of vector quantization that is popular for cluster analysis. It aims to partition a n number of observations into k clusters in which each observation belongs to the cluster with the nearest mean. This results in a partitioning of the data space. The problem is NP-hard. The algorithm is generally composed of two main steps: the assignment step and the update step. In the assignment step, each observation is assigned to the cluster whose mean has the closest distance (usually least squared Euclidean distance). Then in the update step, the new mean is calculated to be centroids of the observations in the new clusters. The algorithm stops (has converged) when the assignments no longer change.

- One scenario when K-Means clustering does not work well is when working with uniform data. In this case, you will still get clusters, but it does not tell you when the data just does not cluster, leading to an dead-end.
- Some of the advantages of K-Means include the fact that it is easy to implement and that with a large number of variables, it can be computationally faster than hierarchical (assuming k is somewhat small). Other advantages include that K-Means can produce higher clusters than other method like hierarchical and the ability for a datapoint to change clusters (to another cluster) after the centroids are recomputed. Some disadvantages could be that K-Means doesn't run well with uniform data and the fact that it is very sensitive to scale. Also, sometimes it can stuck in a local minimum and may struggle with a large amount of clusters.
- One better strategy that could be used to select the K centers would be a density based method where the high density object within the region is considered as the initial cluster center. Other more effective methods could include a random partition-based approach which includes randomly breaking the data into subsets and then performing K-Means on each of the subsets. The results those subset runs are then used as inputs of the K-Means algorithm for the entire data set, being ran the number of subsets from before.