CM 221 Introduction to Bioinformatics

Project Report

Zhiquan(Zac) You
UID: 205667859
03/18/2022

Introduction

K-means clustering is an unsupervised learning algorithm that has been widely used today in the realm of machine learning and data science. This clustering technique gained its popularity due to its simple implementation and efficiency to categorize specific data points into groups even when very little information (also known as label) is given about the data. The algorithm runs iteratively trying to partition the entire data set into k presumably defined clusters. It does this by assigning each data point to a cluster such that the Euclidean distance between the point and the cluster's centroid is minimized at each iteration till convergence.

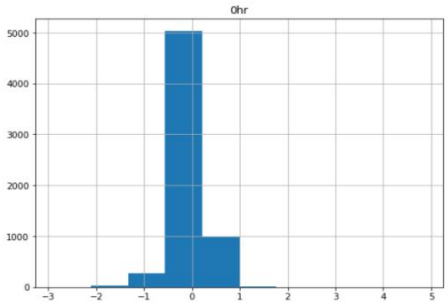
Implementation

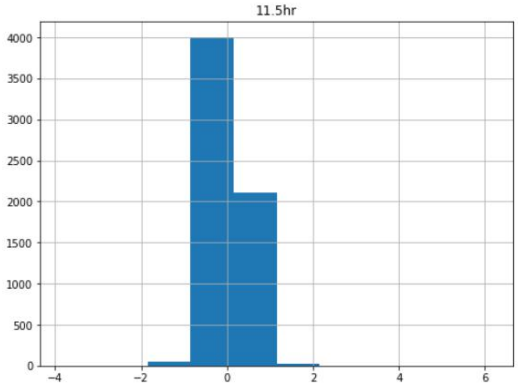
Data pre-processing: The yeast.tsv data set has some missing values on certain rows. Although there are several ways to handle missing values, such as removing the rows, what I did was to fill out the these missing values by the median of the column that missing value belongs to. The motif behind this was that the removal of the rows containing null values might not be a good approach if the missing rows are abundant and removing bit chunk of these data would make our data align to certain trends which is inaccurate.

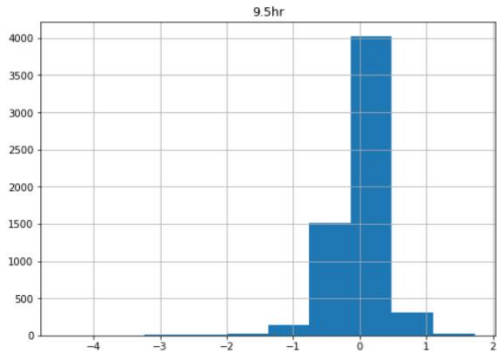
```
df = pd.read_csv('yeast.tsv',sep='\t')
df = df.iloc[: , 1:]

dim = df.columns.values.tolist()
for d in dim:
    df.hist(column=d)
    print(d, ' Skewness:', df[d].skew())
```

As we can see below, most of the data points are skewed, either to the left or right, here I just showed three dimensions, however note that all dimensions have skewed data points. When the data is skewed and not symmetrical, it is a good practice to use the median value for replacing the missing values. And I standardized the data as well.

```
0hr  Skewness: 0.04316209884796835
9.5hr  Skewness: -1.6723128939209213
11.5hr  Skewness: 1.0411750956906514
13.5hr  Skewness: 0.4598085890274028
15.5hr  Skewness: -0.24711966751119882
18.5hr  Skewness: 0.17008252800883317
20.5hr  Skewness: 0.353671541365756
```







Algorithm Coding

i. Initialize the centroids: The very first step of the k-means algorithm is to randomly initialize k centroids to form clusters. As shown below, num_example would be number of rows in our data set, which is 6400. We randomly chose k indices and make points at these indices centroids.

```
def _initialize_centroids_(data, k):
    num_example, _ = data.shape
    rand_idx = np.random.choice(num_example, k, replace=False)
    centroids = data[rand_idx, :]

    return centroids
```

ii. Assigning points to clusters: After we randomly chose k centroids, we assign each point in the data set to its closest centroids. In my code, I calculate the Euclidean distance between each point and each centroids, and append these distances to a distance array. Suppose we have three clusters, then each element of the distance array would be tuples of three. Then we assign each point the cluster index by using np.argmin() function to return the smallest distance to a cluster.

```
def _assign_points_(data, centroids):
    distance_arr = []

    for point in data:
        centr = np.array(centroids)
        distance = np.sqrt(np.sum((point - centr)**2, axis=1))
        distance_arr.append(distance)

    points = np.array([np.argmin(i) for i in distance_arr])

    return points
```

iii. Update the centroids based on the distance: In this step, what I did was I re-initialized an array of centroids pre-filled with 0. Then for points assigned to each cluster in previous step, I calculated the mean of these cluster points and assign these values to new centroids. And I store these new centroids in centroids array and return the new centroids.

```
def _update_centroids_(data, k, points):
    _, features = data.shape
    centroids = np.zeros((k, features))

    for indx in range(k):
        datapoints = data[points == indx]
        new_centroid = np.mean(datapoints, axis=0)
        centroids[indx] = new_centroid

    return centroids
```

iv. Looping through max number of iterations: In this step, we repeat the process of updating the centroids and reassigning points to their nearest clusters till we reach the maximum number of iteration specified by users.

```
for _ in range(max_itr):
    centroids = _update_centroids_(data, k, points)
    points = _assign_points_(data, centroids)
```

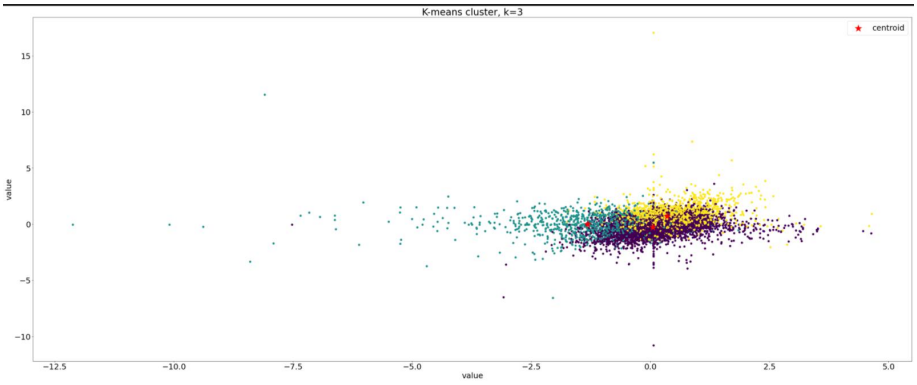
v. Check convergence: I added another stopping criteria within the for loop -- convergence. Once the centroids doesn't change at all, if this happens before the end of the maximum iteration, it would stop the loop.

```
def _is_converged(centroid, updatedCentroids):

    centroid = np.array(centroid)
    updatedCentroids = np.array(updatedCentroids)

    list_of_diff = centroid - updatedCentroids

    return not list_of_diff.any()
```

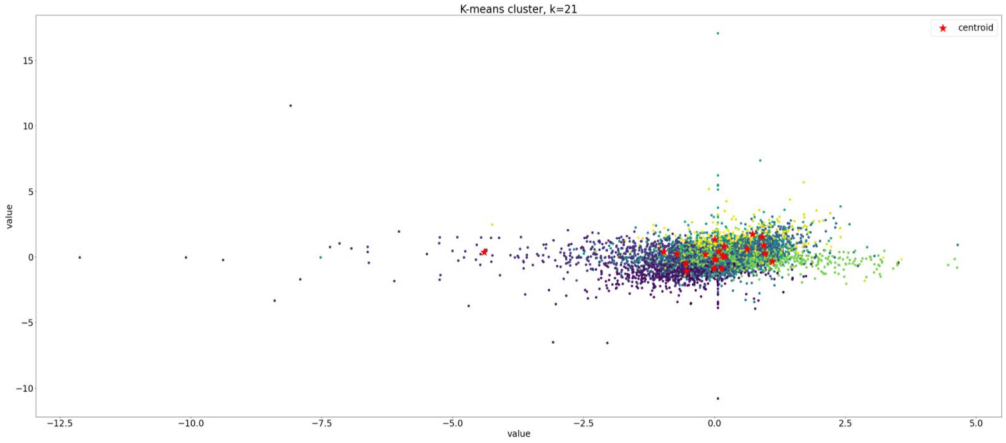
Results

In this section, I will talk about the result of the running algorithm. I've tried several values of k and found some interesting facts about the clustering.

✧ Since there are 7 dimensions of data, I visualized dimension 1 and dimension 2 for the following graphs. What I observed is that the clusters tend to overlap each other somehow. The decision boundaries for each cluster are not that clear.

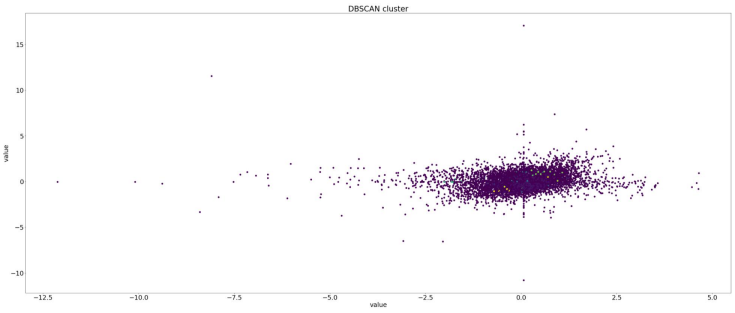


- ✧ There are several outliers in the graph. Initially when k is small, the outliers were assigned evenly to the clusters that are closest to them while from eyeballing the outliers are pretty far away from the main clusters. As the number of k increases, these outliers eventually will get their own cluster centroids and form independent clusters themselves. This may imply that the model is pretty much overfitting. Because the outliers are not supposed to be a cluster. As k increases, the clusters' centers look like they are mainly stayed near each other for the majorities of the points clustered in the middle, with some exceptions for which the centers are formed for these outliers.



I also created a function with sum squares to approximate number of clusters to be used, as explained by Prof. Harold in lecture -- the elbow method. I tested 700 k's and found out that after around 150 clusters the curve don't drop significantly and kind of stay stable. Reference to the code output in my notebook file.

For comparison purposes, I used DBSCAN clustering algorithm from sklearn to cluster the dataset. Unlike k-means, DBSCAN does not require us to specifiy the number of clusters in the data a priori. However, the result didn't turn out well. As we can observe from the cluster below, DBSCAN returns a major cluster with some small dots(small clusters within the major cluster). The reason behind it was that DBSCAN is density based clustering algorithm, meaning that it would do well in terms of seeking areas in the data that have high density of observations and compare to the area of data that are not very dense with observations. So in our case, since we visualized dimension 1 and dimension 2 only and data from these two dimensions heavily overlap or are very close to each other, DBSCAN would just cluster one big cluster on the high density area. More interesting to note is that DBSCAN also clusters these outliers that spread out in the left into the big purple cluster. The reason why k-means form clusters on the densed data sets is that k-means are based on Euclidean distances as its measurements, so it does not take density into account, just the distances between points and centroids.



In summary, k-means algorithm that I implemented forms clusters on the densed data set, with the increase of k components, the cluster seems to overlap somehow, and the outliers gradually are assigned a centroid to form a cluster which indicates the possibility of overfitting. Also I used PCA as a way to reduce dimensionality of our data set (just for testing purposes) and ran the implemented k-means on the reduced components. I do observe that for the first few k's, the clusters are easily distinguishable, meaning that the decision boundary is sort of clear.