AN IMPLEMENTATION OF k-MEANS CLUSTERING

BRENDAN WHITAKER

ABSTRACT. We implement from scratch the k-means clustering algorithm and apply it to the `TwoDimHard` dataset which contains two numeric independent variables with 4 true, slightly overlapping clusters, and 400 examples. The dataset was sourced from the instructor for the course, Jason Van Hulse. We design a program which accepts as a parameter the number of clusters k, specified by the user. We implement the standard Euclidean distance measure for use in the k-means algorithm. The output of the program will consist of two columns - (1) the row ID, and (2) the cluster that each record belongs to, as determined by the clustering method. We also compute measures of cohesion and separation of the clusters to evaluate our algorithm.

CONTENTS

1. DATA PREPROCESSING AND IMPLEMENTATION

This program was designed in Google Colab, which is very similar to Jupyter notebook. It runs in Python 2. We initially had difficulty learning how to upload the data file (csv) since unlike Jupyter Notebook, Colab runs in the cloud and thus does not interface with a hard drive directory. Instead, it accesses any files in the user's Google Drive, but we're also able to write code to implement a direct upload to the Colab workspace, so that's what was done to load the CSV. It's important that the file-name the user uploads matches the original file-name "TwoDimHard.csv", since otherwise the rest of the code won't function correctly.

Date: SP18.

The first thing we did is implement a normalization of the dataset, so we set the mean to 0 and the standard deviation to 1. However, we found that this did not significantly improve the performance of the clustering algorithm, so it was commented out in the final program.

Next, we implemented a GUI for the user of the program to select the number of clusters for the algorithm, i.e. to select the value of k. This keeps the entire program in full generality so that it can be used for an arbitrary dataset where you may want more or less than 4 clusters with minimal modification.

After that, we implemented a short python function for Euclidean distance. Recall that Euclidean distance is defined for two vector $\mathbf{x} = (x_1, ..., x_n), \mathbf{y} = (y_1, ..., y_n)$ in $\mathbb{R}^n$ as:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (y_i - x_i)^2}. \tag{1}$$

We ran into an issue during this implementation that amounted to the "**" operation used for exponentiation not working for fractions, and so instead we had to use the `math.sqrt()` function. This issue resulted in an raw error count of over 200, which is quite bad since the dataset is only 400 rows long. But once we had finished fixing the Euclidean distance implementation, this dropped down to less than 100.

Now comes the implementation of the k-means algorithm itself. This is quite simple to write in pseudocode, but recomputing the centroids and getting the loop condition to update correctly proved to be quite difficult. The centroid computing function simply loops over i up to k, and for each row in the database, it computes the mean using a running sum of the coordinates, and a count of the number of relevant datapoints. It then adds the new centroid to an array which is returned at the end.

For the loop, we loop over all rows to compute distances, and use a running minimum to find the correct cluster assignment. During this same loop, we also check if this new assignment is the same as the previous one. If this is true for all rows, then we terminate the loop, as this means that it has converged to the optimal clusters for these parameters.

2. Post processing and results

This was where the lion's share of our bugs occurred. Our implementation of the algorithm was not guaranteed to use the same names for cluster labels as the true clusters given to us in the dataset. This proved to be quite a problem, since it is impossible to get an accurate measure of how closely the algorithm followed the true labels if the names don't match up. The way we solved this is through a relabeling algorithm, which uses frequency analysis on each cluster to determine out of all k possible relabellings for this cluster, which provides the greatest gain in raw accuracy. So looping over each cluster, we get permutation of the clusters, which we then use a loop over all rows to implement. The result is a final output `DataFrame` whose cluster labels match the true labels of the dataset as much as possible. At then end of all this, we managed to yield a raw error count of 21 out of 400 rows, which is quite an improvement over the initial tests, which had error rates in the hundreds.

We computed the cluster SSB (separation) array for the true clusters:

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \hline 61.7 & 34.7 & 45.4 & 49.5 \end{array}, \tag{2}$$

And the cluster SSE (cohesion) array for the true clusters:

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \hline 0.313 & 0.903 & 2.430 & 1.911 \end{array}, \tag{3}$$

And the total SSE for the true clusters, which was 5.556.

We also computed the cluster SSB (separation) array for the newly generated clusters (the ones generated by our algorithm):

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \hline 60.6 & 34.1 & 48.9 & 53.2 \end{array}, \tag{4}$$

As well as the cluster SSE (cohesion) array for the new clusters:

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \hline 0.500 & 1.076 & 1.845 & 1.471 \end{array}, \tag{5}$$

And the total SSE for the new clusters, which was 4.892. What's very interesting about these results is that the total sum of squared error (SSE) for the generated clusters from our new homebrew algorithm is actually lower than that of the true clusters. This reflects the truth that the distribution we are trying to model when doing data science is not always "ideal" or "nice", whereas our models tend to assume that it is. This is reflected in the side-by-side scatterplots of the two cluster assignments:

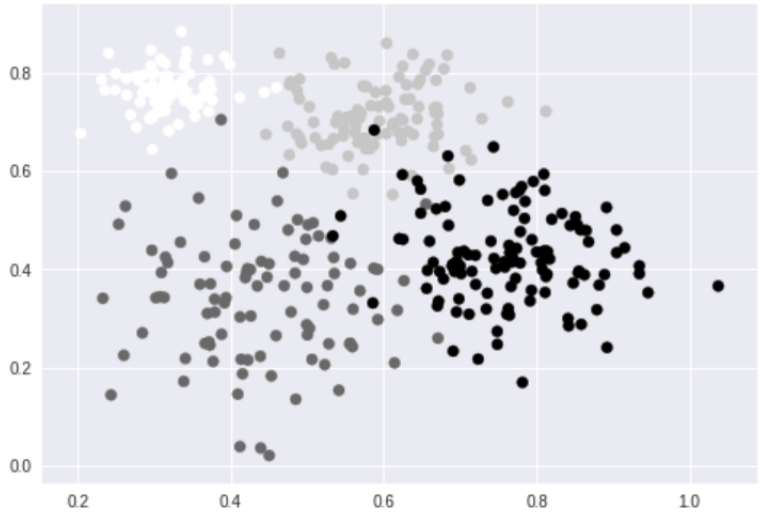

FIGURE 1. true clusters.

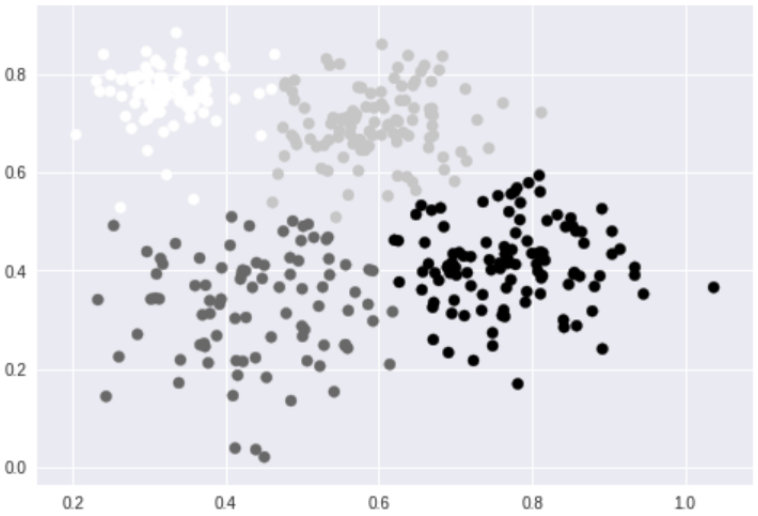

FIGURE 2. new clusters.

Observe that in the true clusters, there is some noise, like the lone black point in the sea of light gray points, and the few gray points which have encroached on the white cluster. But in the new clusters, none of this noise is present, the clustering is "better", but it doesn't fit the true distribution perfectly.

We can see these small discrepancies more clearly in the cross-tabulation matrix:

	1	2	3	4
1	89	0	0	0
2	2	98	0	0
3	4	2	88	3
4	0	8	2	104

. (6)

3. The $k = 3$ case.

We now move on to the discussion of the results when we reduce the number of clusters down to $k = 3$ in our implementation of k-means. Luckily, since the code was very generalizable, it was fairly simple to implement this change. We simply input the new value for k and ran each code block in the Colab notebook in order down the file, and since we already wrote the code to compute the SSE array, total SSE, and SSB array, computing these for $k = 3$ was very quick. We give these results and compare to the case where $k = 4$. Our SSB (cohesion) array is:

1	2	3
52.8	27.8	36.0

, (7)

And our cluster SSE (cohesion) array for the new $k = 3$ clusters is:

1	2	3
0.870	1.667	7.855

, (8)

And the total SSE for the new clusters, which was 10.393. The SSB values are quite similar to the case where $k = 4$, and this makes sense, since removing a single cluster shouldn't affect the inter-cluster distances all too much. We would expect a great increase in inter-cluster distance (separation) if we went from a very high cluster value of k, 50 for example, to a lower cluster value of say $k = 4$. The SSE array is more telling, We had values hovering around 1-2 in the $k = 4$ case, but in the $k = 3$ case we had one cluster which read more than 7, which is much higher than the highest entry from the $k = 4$ array. Similarly, the total SSE for the $k = 3$ case was about double that of the $k = 4$ case, at around 10 compared to the 4.9 value of $k = 3$.

4. Off-the-shelf algorithms

We used the `sklearn` k-means off-the-shelf clustering algorithm to compare my implementation. The code for the implementation is as follows:

```
# Convert DataFrame to matrix
mat = raw.as_matrix()
# Using sklearn
km = sklearn.cluster.KMeans(n_clusters=4)
km.fit(mat)
# Get cluster assignment labels
labels = km.labels_
# Format results as a DataFrame
results = pd.DataFrame([raw.index,labels]).T
print results
```

As seen above, there are no model parameters to tweak other than the value of k, which was set to 4 to make it easy to compare this implementation with the one made in this project. As we did with the homebrew algorithm, we compute the SSB and SSE arrays and the total SSE for this implementation to see how it compares: We have the SSB:

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \hline 42.0 & 56.2 & 51.2 & 32.0 \end{array}, \tag{9}$$

As well as the cluster SSE (cohesion) array for the new clusters:

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \hline 3.694 & 1.168 & 1.666 & 2.306 \end{array} \tag{10}$$

And the total SSE for the new clusters, which was 8.835. What's curious about this is that the error metrics are actually lower by a significant measure than the ones we computed for our new homemade k-means, although they are still quite close in the grand-scheme of things. They could be using a different distance metric other than Euclidean, which is probably the most likely reason for the difference. It could also be rounding error, or they could be using a slightly different termination criterion than we are. However, since our accuracy isn't lower than theirs, and this is a widely used implementation, we can safely assume that our algorithm works sufficiently well.