Analysis of the k-Means Data-Clustering Algorithm

Mauricio Tedeschi
CS2810 - Mathematics of Data Models, Fall 2021

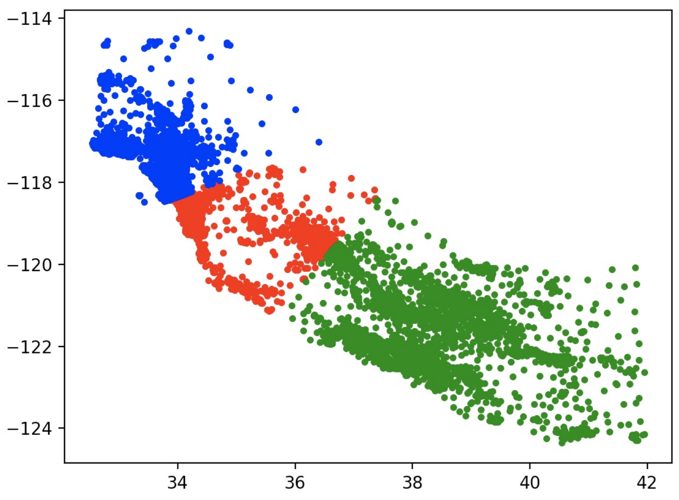
December 14, 2021

Introduction

In data analysis, it may be in one's interest to find relationships between data points with respect to their neighborhood. There are many instances where scatter-plots of data reveal points in "clumps," which may look like stellar clusters in galaxies. These dense clusters may give off interesting conclusions about the nature behind the data points. A relevant example of this could arise from mapping out locations of the COVID-19 Omicron Variant by longitude and latitude. When observing where more dense clusters of points appear, it may suggest to the viewer where more severe outbreaks are coming from. People may utilize computer programs to distinguish between clusters of data to find relationships between specific groups of points. Many such algorithms were developed to calculate groups of points to make the job of searching for clusters both faster than accurate than the process of doing so by hand. In this report, we will analyze the structure behind and the accuracy of the k-Means clustering algorithm.

Structure

The k-Means algorithm serves to approximate clusters of points in any n-dimensional space. A benefit of this algorithm is that it is one of the easiest and most intuitive data-clustering algorithms to implement. The algorithms enables for data points to be grouped into a user-defined number of clusters, k.



k-means cluster visualization for a set with 3 clusters, on the data points for (longitude, latitude) of "houses," extracted from *housing.csv*.

The figure above illustrates how the k-Means method would manipulate data points to point to a certain cluster as indicated by their color. We will now examine the design and flow of the source code of my implementation running behind the scene of the figure.

```python
import numpy as np
import random
import matplotlib.pyplot as plt


"""
Mauricio Tedeschi
CS2810 - Mathematics of Data Models
14 December 2021

k-means clustering implementation with Jaccard score analysis
"""

class Point:
    def __init__(self, pos, clusterID):
        self.pos = pos
        self.clusterID = clusterID

    def get_dist(self, other):
        r = self.pos - other.pos
        return np.sqrt(np.dot(r, r))

    def get_ID_from_nearest_neighbor(self, M):
        min_d = np.Inf
        for i in range(len(M)):
            d = self.get_dist(M[i])
            if d < min_d:
                min_d = d
                self.clusterID = i

    def to_string(self):
        return "((" + str(self.pos) + ", " + str(self.clusterID) + ")"

# dataset is a list of (x1, x2, ..., x_N), the number of dimensions
# is number of dimensions in every vector element

# returns list of points labeled with a cluster ID: C = 0...(k-1)
def k_means(file_data, title_args, k):
    dataset = read_data_from_file(file_data, title_args)
    means = random.sample(dataset, k)
    previous_means = []
    while previous_means != means:
        previous_means = means
        for x in dataset:
            x.get_ID_from_nearest_neighbor(means)
        clusters = [[x for x in dataset if x.clusterID == i] for i in
    range(k)]
        for i in range(k):
            new_centroid_pos = sum([x.pos for x in clusters[i]]) / len(
    clusters[i])
            means[i] = Point(new_centroid_pos, i)
    return clusters

# title_args is the list of column titles for the parameters
# of interest
def read_data_from_file(filename, title_args):
    file = open(filename)
    is_header = True
    column_headers = []
    output_list = []
    for line in file:
        element = np.array([])
        line_sep = line.split(",")
        if is_header:
            column_headers = line_sep
            is_header = False
        else:
            for i in range(len(line_sep)):
                if column_headers[i] in title_args:
                    element = np.append(element, float(line_sep[i]))
            output_list.append(Point(element, -1))
    return output_list

def get_jaccard(l1, l2):
    s1 = set([tuple(x.pos) for x in l1])
    s2 = set([tuple(x.pos) for x in l2])
    return len(s1 & s2) / len(s1 | s2)

# Although k-means can work around any number of dimensions, the plotter
# will only display the first two
def display_k_means_plot(dataset, k):
    color_modes = ["r.", "g.", "b.", "m.", "k.", "y."]
    for i in range(min(len(color_modes), k)):
        plt.plot([x.pos[0] for x in dataset[i]], [x.pos[1] for x in dataset[i]], color_modes[i])

def k_means_analysis(file, title_args, k):
    output = k_means(file, title_args, k)
    other_out = k_means(file, title_args, k)

    for i in range(k):
        for j in range(k):
            print("Cluster", str(i), "with", "Cluster", str(j), ":", get_jaccard(output[i], other_out[j]))
    print()

    display_k_means_plot(output, k)
    plt.show()
    display_k_means_plot(other_out, k)
    plt.show()

if __name__ == "__main__":
    k_means_analysis("housing.csv", ["Longitude", "Latitude"], 4)
    k_means_analysis("abalone.csv", ["Diameter", "Height"], 4)
    k_means_analysis("autos.csv", ["width", "height"], 4)
```

The program makes use of the `numpy` in order to perform more efficient calculations with vector quantities. The module `random` allows for the program to take a random sample of k initial centroid points to preface the iterative clustering process. The module `matplotlib.pyplot` allows for a convenient display of the clustered data points to be processed.

The first segment of the code introduces the Point class. The k-means would extract vector quantities from stored data sets, subsequently to be paired with a unique cluster label, hence the motivation to incorporate an object to store both *position* and *cluster ID* fields. Point also contains methods to find the distance between itself and another Point, and one to assign a *cluster ID* based on the closest point out of a set of points, M. These implementations will be useful for the core of the k-means procedure.

The Core Method

The k-means procedure shown in lines 38–51 is based on the following design:

1. The user sets initial parameters for the number of clusters k and the preset data file and quantities of interest to be extracted from the data.

2. Parse the data from a supplied file into a compatible data structure, `dataset`. Since there are various quantitative and categorical properties in each item in the data file, only a select sample of properties are considered as specified by the user. For instance, we take only the “Longitude” and “Latitude” quantities from the items in the `housing.csv` file. Note that `dataset` is a constructed list of Points, so we initialize the `clusterID` of each Point to -1 to indicate that no cluster is yet identified for said Point.

3. Take a random sample of points $\{\mathbf{c}_i\},\ i \in \{0, ..., k-1\}$ from the data set. These will represent the centroids for each cluster before the iterative process begins.

4. Keep repeating this step until the cluster centroids cease to shift positions from iteration to iteration. For every point $\mathbf{x} \in$ `dataset`, set its `clusterID` to the argument i of the centroid $\mathbf{c}$ whose distance from $\mathbf{x}$ is the smallest compared to other centroids:

$$\text{clusterID}(\mathbf{x}) = \underset{i}{\text{argmin}}||\mathbf{x} - \mathbf{c}_i||\ \forall\ \mathbf{x} \in \texttt{dataset}.$$

   We then take each cluster, C_i, and calculate a new centroid for it by finding the average value:

$$\mathbf{c}'_i = \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}.$$

   Note that this step suggests that all points carry equal weight. That concludes the iterative process.

5. Once the centroid values finally converge into fixed points, we can then output the modified points (with new `clusterID`’s).

The methods below the k-means method are supplemental for the program, such as reading data points from files, computing the Jaccard similarity between two sets of points, and displaying plots and test statistics.

Test Files

This program has been tested with data set files including the following:

- housing.csv - https://www.kaggle.com/ryanholbrook/clustering-with-k-means/data?select=housing.csv
- autos.csv - https://www.kaggle.com/ryanholbrook/clustering-with-k-means/data?select=autos.csv
- abalone.csv - https://www.kaggle.com/ryanholbrook/clustering-with-k-means/data?select=abalone.csv

Results

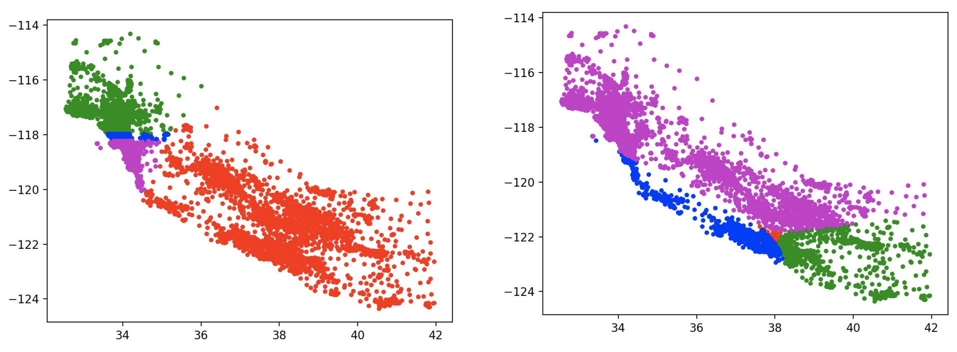
The data visualizations and Jaccard similarity metric are used to assess the accuracy of my implementation of the k-means algorithm.

To briefly explain Jaccard similarity, it is a metric comparing how "similar" two sets A and B are to each other. Its formula is $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$.

For each file, I would run two executions of the k-means procedure on the file and compare the results of the clusters of each computation via Jaccard similarity. We will soon observe that k-means is highly sensitive to the initialized centroid values, as they are randomized. To address this, I made sure to calculate the Jaccard of one cluster C_i from the first execution to all the clusters C_j from the second execution. For all the tests, I set k to be 4.

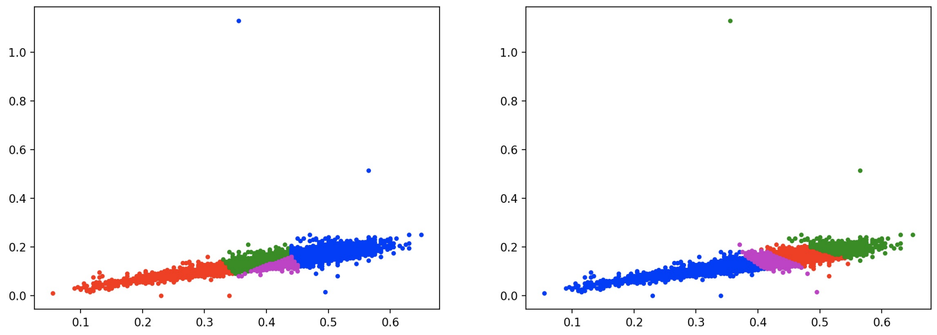
Statistics

Test 1: `file: housing.csv,` $(\mathtt{x}_1, \mathtt{x}_2)$`: ('Longitude', 'Latitude')`



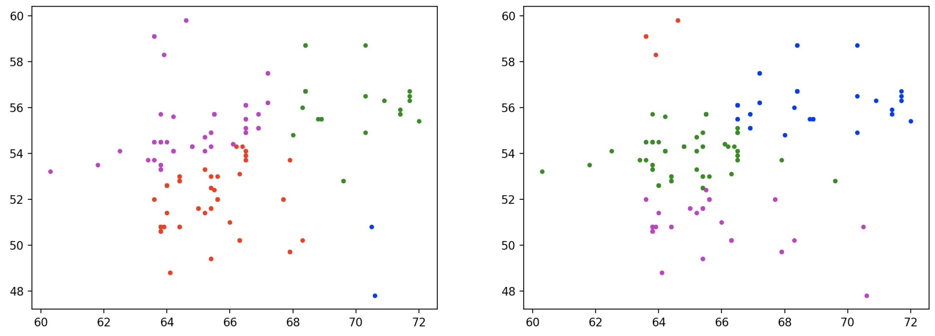
```
Best Jaccard scores: {0.3298, 0.3717, 0.1247, 0.1661}
```

Test 2: `file: abalone.csv`, $(\mathtt{x}_1, \mathtt{x}_2)$: `('Diameter', 'Height')`



`Best Jaccard scores:` $\{0.6861,\ 0.4276,\ 0.0281,\ 0.3547\}$

Test 3: `file: autos.csv`, $(\mathtt{x}_1, \mathtt{x}_2)$: `('width', 'height')`



`Best Jaccard scores:` $\{0.5294,\ 0.6956,\ 0.1000,\ 0.4783\}$

Observations

In the first k-means execution on Test 1 (left plot), the largest cluster spans across roughly 70% of the points from the right. But in the second execution (right plot), the supercluster spans along the upper region of the point distribution. The Jaccard scores fall in a low range from 0.1247 to 0.3298.

Test 2 seems to have a bit more improvement in its end result compared to Test 1. The clusters *nearly* follow a similar pattern, where the (seemingly) largest cluster is on the left wing of the distribution. The two middle clusters are in slightly matching positions in both plots. Better yet, the Jaccard scores raised up to a range peaking at 0.6861. It seems that the minimax Jaccard score was a rather measly 0.0281.

Test 3 seems to yield rather decent results too. The clusters seem to fall on top of similar groups of points in both plots, with the exception of the fact that there are two outlying blue points in the left plot when there is an overarching cluster in the right plot, and that there are three outlying red points in the right plot when the left plot has a cluster occupying a whole upper-left region. The Jaccard scores are by far the best in this test: a set ranging from 0.1 to 0.6956.

Conclusions

The k-means algorithm is a relatively interesting algorithm. It has a rather simple theory behind it compared to most other data-mining algorithms, so it is easier to implement than most. However, I would be cautious about using this algorithm for clusters that are not easy to discern. If I were to test data sets where all the points are evenly distributed within a regular enclosed perimeter, I would expect for there to be much smaller Jaccard similarity scores for those tests.

Recall how poor the algorithm is for yielding fixed results independent of the number of times executed. The first aspect of k-means that seems to reflect its flaws the most is how volatile it is at assigning clusters. While the algorithm is good at converging its values in a fixed number of iterations, its foundations lie on the fact that if the user doesn't already do it, the program will randomly select a sample of points to serve as cluster centroids.

Should there be any modifications to this algorithm, I would add a feature involving a user interface. A user can pinpoint locations on a point distribution plot for where they wish for the initial cluster centroid values to be at. This could perhaps help significantly reduce the margin of error of the algorithm's calculations and the variability between runs of the algorithm on a particular data set, since there would be an influence from the user's frame of reference. Especially for datasets where clusters are pretty obvious to make out, people could easily assist the program to calculate the k means the way it's intended.