An Analysis on Clustering based on Kruskal's Algorithm

Xiaobo Qian, Yangli Liu, Lu Wang, Jieqi Yang

Abstract

In this project, we were exploring the performance of Kruskal's Algorithm on clustering. We tested the algorithm on four different datasets, and by comparing with K-means clustering, we found that Kruskal's algorithm clustering showed better performance than K-means clustering. Further, we also tried to make some improvements to the clustering approach. By comparing the results before and after the improvements, we found there was no obvious change on the final formed clusters if we changed the function of calculating the distance. Besides, we also tried to improve the clustering approach by including the density property of datasets into our calculation.

Project Introduction

- The topic of our project

Many things around us can be classified as "this and that" and we make choices every single day, from which type of pizza we want to order to what type of job we might want to look for. Sometimes the choices are clear, but sometimes we don't have predefined choices at the beginning, rather, we derive those choices based on particular hidden patterns. This process is known as clustering, a

process during which we dig out the underlying patterns of sample data and partition a given dataset into homogeneous groups.

Clustering is one of the major data analysis tools in the field of data mining and it is also a Machine Learning technique that involves the grouping of different data points. Given a set of data points, we can use a clustering algorithm to classify each data point into a specific group. Theoretically, data points that belong to the same group should have similar features, while data points in different groups should have significantly dissimilar properties. By conducting clustering analysis, we can gain valuable insights from the given data by seeing what groups the data points fall into when we apply a specific clustering algorithm. In practice, clustering helps identify the meaningfulness of data. For example, in the medical field, researchers apply clustering to gene expression experiments; In the business world, people use clustering for customer segmentation, which businesses can then utilize to create targeted advertising campaigns.

- What we are going to explore in our project

Despite the fact that there are different types of clustering algorithms, each of which offers a different approach to the challenge of discovering natural groups in data, most of them have limitations in meeting their robustness and quality for clustering. In this project, we are going to further explore Kruskal-based clustering and K-Means clustering by diving into the theories behind how the two algorithms work and comparing their clustering performances directly. After that, we will discuss some potential improvements for our clustering approach.

- The project motivation of our teammates
    - Yangli Liu: With a background in Fine Art, I am mostly interested in what kind of algorithms that I learned from this course can be used to help train the computer to better recognize, understand, and even

interpret the visual word. I came across several applications that are developed by clustering techniques. One of them is doing image classification that can group the same or similar shapes of elements into the same cluster.

- Jieqi Yang: In course 5800, we learned how to find the Minimum Spanning Tree in a graph by Kruskal's algorithm. I am very interested in the application of this algorithm in real life. After doing some research, I know this algorithm can be used in clustering analysis. I had never heard of this application before, so I decided to do some research with my teammates on this topic and figure out how Kruskal's algorithm works on clustering. Besides, I also hope to dive a little deeper to see what are the pros and cons of Kruskal's algorithm compared with other algorithms when dealing with clustering problems.
- Lu Wang: I come from a mixed background of business and policy prior to this CS journey, and am very interested in graph theory, which is a large area of research for mathematicians and computer scientists. In graph theory, the shortest path problem is a very important topic in the business sector and also in the policy world. After taking CS5800, we learned that the Kruskal algorithm is one of the algorithms used in solving minimum spanning tree problems and obtaining the optimal path.I became very interested in this topic afterwards and was intrigued to know more about it. In this group project, my teammates and I will be further exploring clustering analysis based on Kruskal's algorithm and discussing the implementation and potential areas of improvement of this topic.
- Xiaobo Qian: In a previous machine learning course, I learned some popular clustering algorithms, such as k-means, density-based clustering, etc. In cs 5800, I learned that Kruskal Algorithm can be

also applied to solving clustering problems. However, it is rarely used today. Of course it has some drawbacks, while I am interested in how it can be improved with other techniques in clustering and the process of exploring such topics can deepen my understanding in clustering problems.

- The data source

We chose four public datasets as our test data. The data point distribution of each dataset was shown as follows.

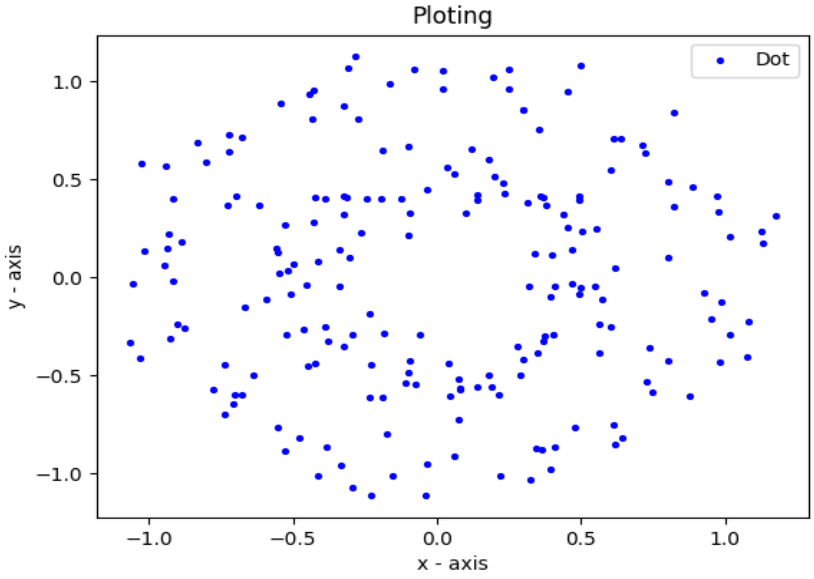

Fig.1 Two Circles Data

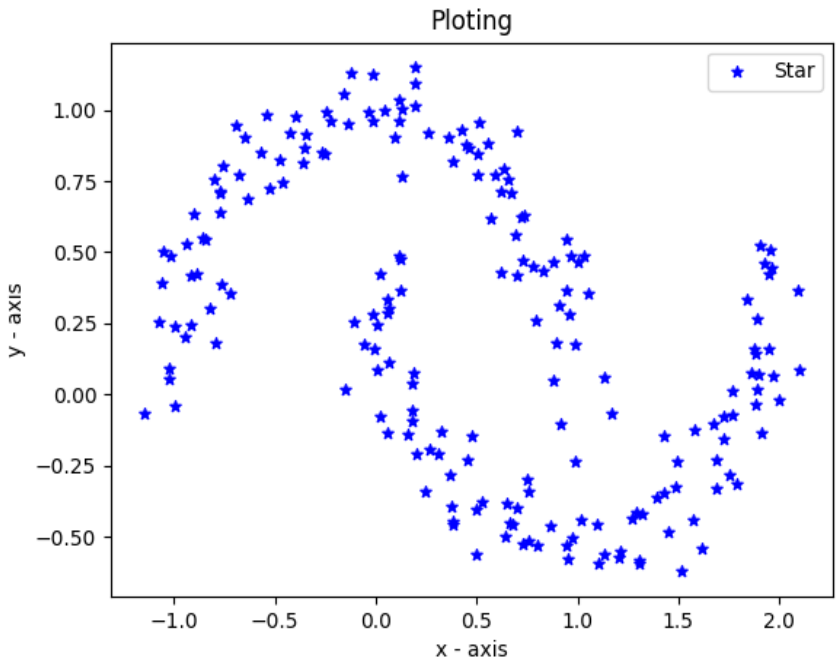

Fig.2 Two Moons Data

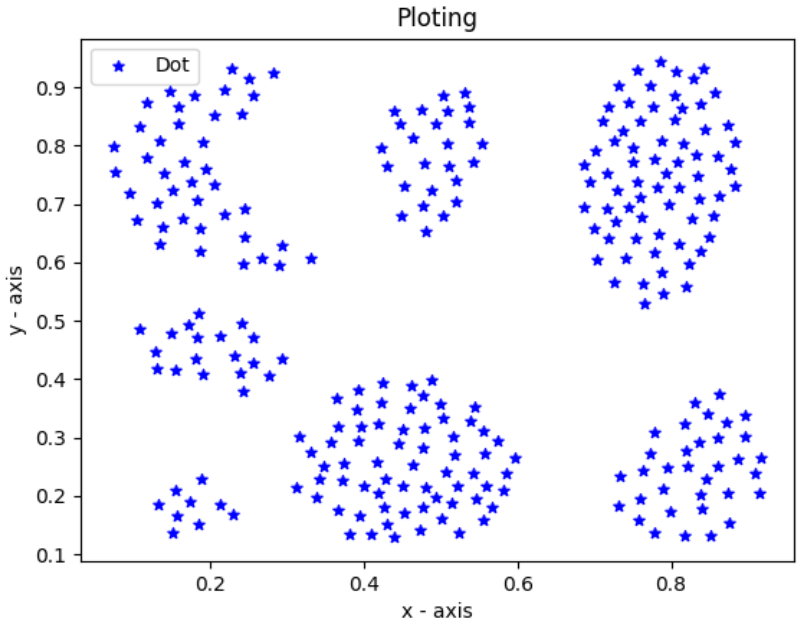

Fig.3 Normal Data

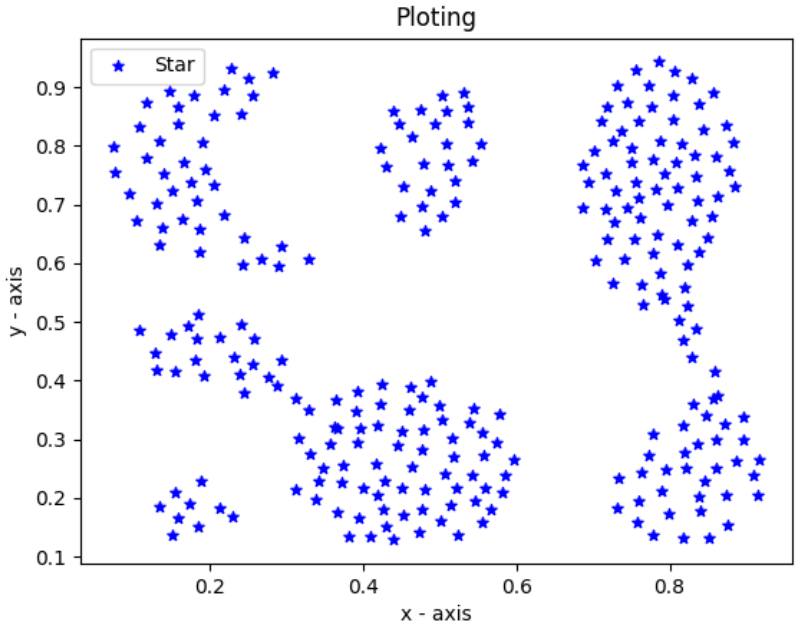

Fig.4 Single Link Data

Our whole project is based on the above datasets. First, we clustered each dataset by Kruskal's Algorithm. Then in order to check the performance of Kruskal's Algorithm, we also clustered the same datasets by K-means approach. Finally, we tried to make some improvements to our Kruskal's algorithm clustering based on distance and density of data points.

Clustering based on the Kruskal's Algorithm

1. Introduction to the Kruskal's Algorithm

As we have learned in this course, Kruskal's algorithm is one of the most popular greedy algorithms for finding the minimum spanning tree (MST) of an undirected graph. Using Kruskal's algorithm, we will be able to define a group of edges from

the designated graph that connects all the vertices with the minimum total edge weight and without forming any cycles.

2. How does Kruskal's Algorithm work

When it comes to the technical details of Kruskal's algorithm, we need to sort all the edges in non-decreasing order based on their weights. The straightforward way is to create an empty set T and then add the edge from the sorted edge array into T as long as it does not form a cycle with edges that are already in the set. The pseudocode for Kruskal's algorithm is shown as following:

```
-- Sort edges in ascending order of edge weights
-- T = empty
-- for i = 1 to the length of the array
-- Let e = E[i]
-- If e does not make cycles in T:
   -- Add e to T repeatedly
```

In order to speed up the process of checking the cycle's existence, we need to create a data structure called Union-Find. This structure is also called merge-find as it stores a collection of non-overlapping sets. In the actual implementation, we see that Find(group, x) operation tells if value x has the same parent of the group, and Union(group1, group2) function makes a union set from the two individual sets group1 and group2. After setting the useful utility function Find and Union, we can move on to pick the smallest edge and increment the index for the next iteration. If the Find function tells us that including this edge does not cause a cycle, we include it in the result array and increment the index of the result array by one. Otherwise, we discard the edge.

3. Clustering based on Kruskal's Algorithm

Just for the purpose of analyzing this big clustering topic, we have adapted what we learned from Kruskal's algorithm of finding MST into searching the largest possible value of the distance between any two data points from different subsets, in other words, the spacing of the clustering. What is worth mentioning in writing the actual code of this K-clustering algorithm is that each data point is a node with properties of parent and rank, and each node is a single cluster at the beginning. We maintain clusters as a set of connected components of a graph. After iteratively combining the clusters containing the two closest nodes by adding an edge between them, we will be able to draw the clusters in a forest-like way. The algorithm will stop when there are k clusters that k numbers are pre-selected by us. It will return the actual decimal numbers of the spacing of the clustering. We attached the image below(Fig.5).

```
The largest possible value of distance between any two
    points from different subsets is at least 0.2190018402

The largest possible value of distance between any two
    points from different subsets is at least 0.2063165400

The largest possible value of distance between any two
    points from different subsets is at least 0.0943055084

The largest possible value of distance between any two
    points from different subsets is at least 0.0945741774
```

Fig.5 The Spacing of Clustering

We have tried four different shapes of the unlabeled data sets, we intentionally used red color to highlight the resulting edge. The clustering results for the four datasets as follows:

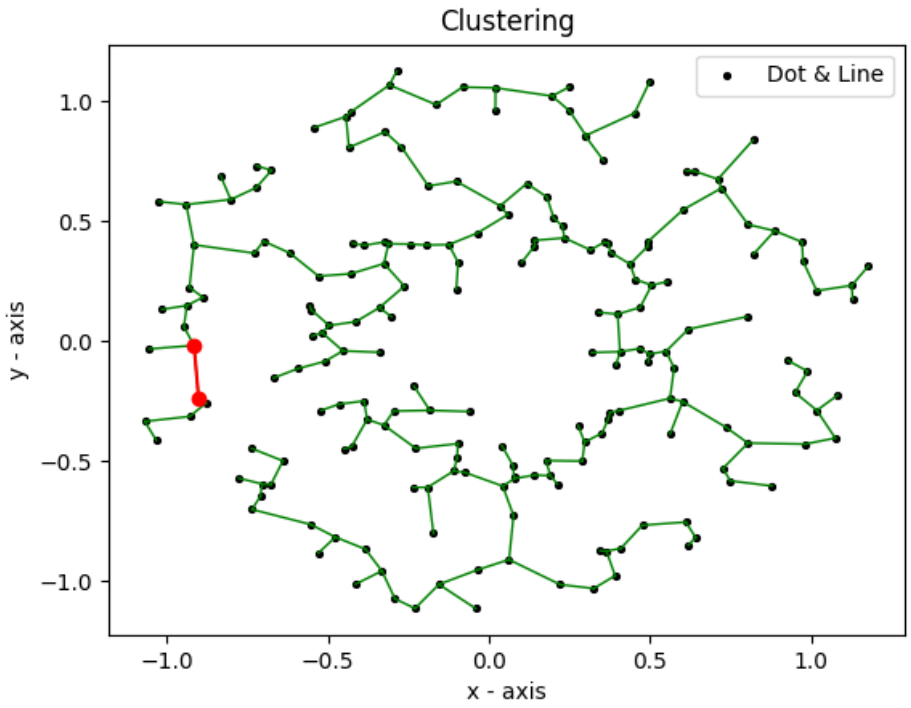

Fig.6 Two circles by Kruskal's Algorithm

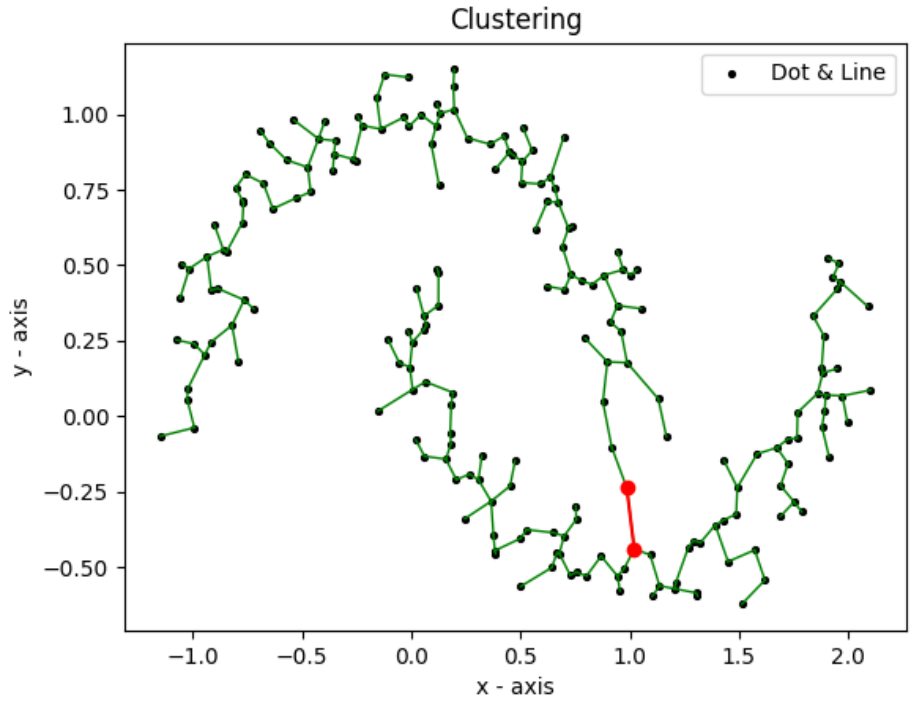

Fig.7 Two moons by Kruskal's Algorithm

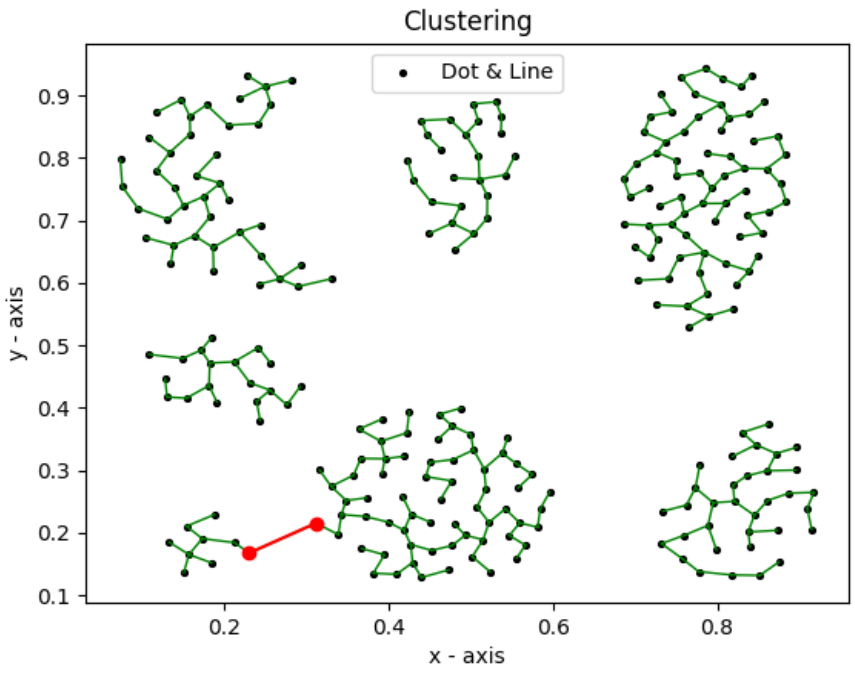

Fig.8 Normal data by Kruskal's Algorithm

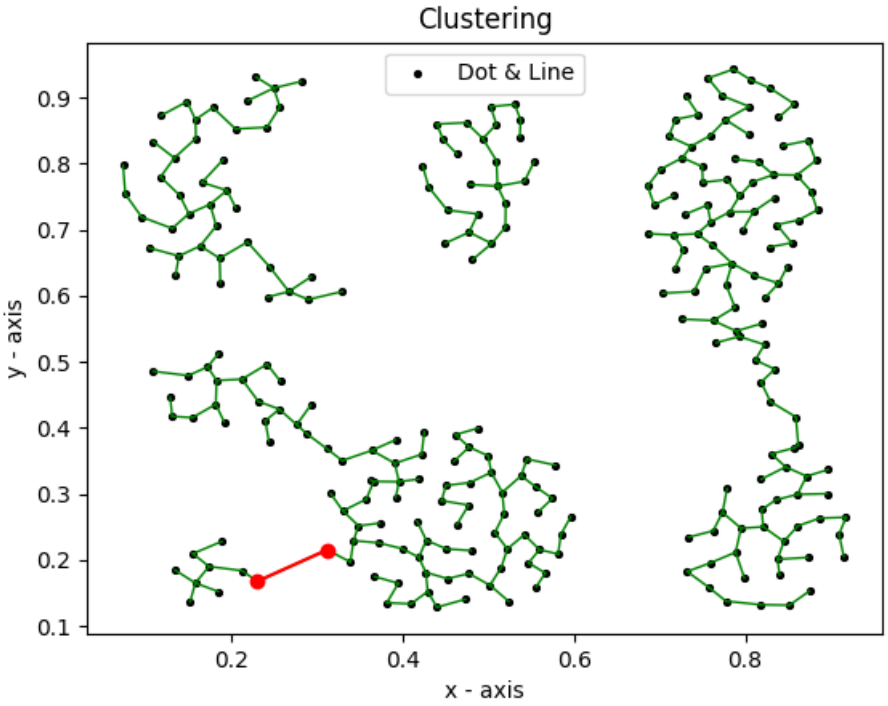

Fig.9 Single Link by Kruskal Algorithm

K-means clustering

We implemented K-means clustering on the same datasets and compared the output of K-means clustering with that of the Kruskal's algorithm-based clustering.

1. Brief introduction to K-means clustering

K-means is an introductory algorithm to clustering techniques and it is widely used in unsupervised learning problems. K represents the number of clusters we are going to classify our data points into.

2. How does K-means clustering work

Simply speaking, K-means clustering repeatedly calculates and finds K centroids and group points with their nearest centroid. If we specified the maximum number of iterations, say m, the algorithm will stop after m iterations. If we don't specify the maximum number of iteration, the algorithm will stop if the newly formed centroids don't change any more or the points stay in the same cluster without any change. In our project, we didn't specify the maximum number of iterations.

The steps of K-means clustering:

1) We specify the value of K, which means we need to tell the algorithm how many clusters should it classify the dataset into at the end;
2) Randomly pick K points from the dataset as the initial centroids;
3) Among all other points in the dataset, calculate the distance between every point to each of the centroids in the second step. Group each point to its nearest centroid together.
4) After the third step, we got K clusters. Calculate the centroid again for each cluster. With the new centroid, we repeat steps 3 and 4.

5) The algorithm will stop once either the newly formed centroids don't change or the points remain in the same cluster.

3. K-means clustering pseudocode

```
Import dataset D;

Specify the value of K and/or the maximum number of iteration if needed;

Choose the initial centroids c_1, c_2, ... c_k randomly;

K-means(D, K):
  for each data point d_i:
    find its nearest centroid c_n among c_1, ... c_k;
    group d_i with c_n into a cluster;

  for each cluster j = 1..k
    Calculate and find its new centroid = mean of all points assigned to cluster j

  Repeat the above two loops until convergence or until reach to the maximum number of
iterations
```

4. The output of the K-means clustering

We implemented the K-means clustering in Python by specifying the value of K is 2. With the help of the numpy package and the plotly library in Python, we got the clustering result for the same datasets as follows:

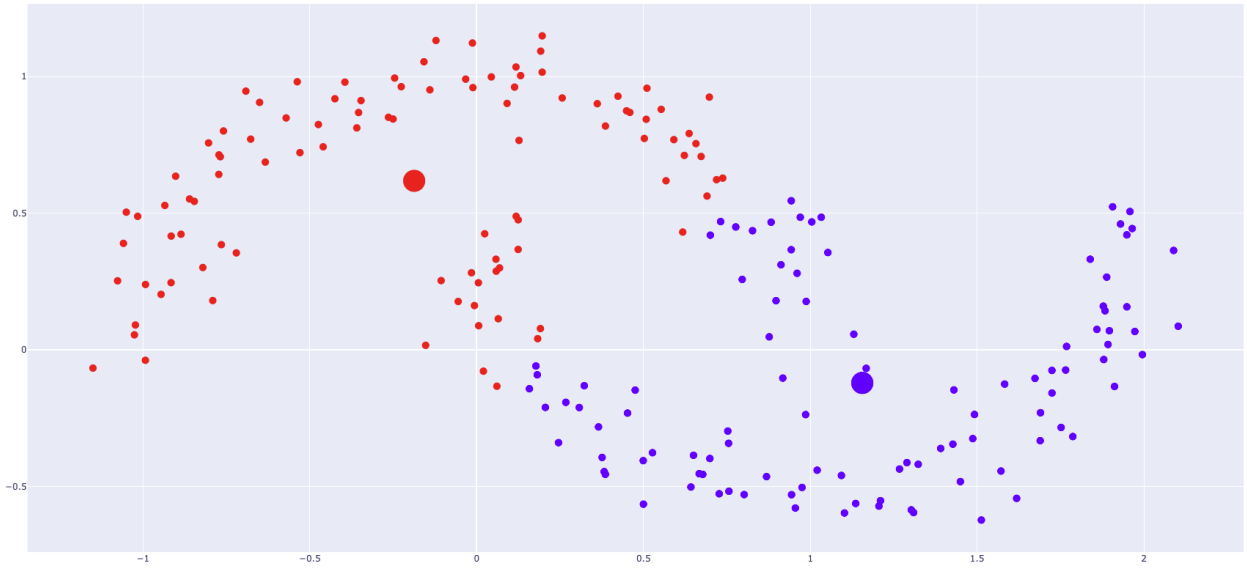

Fig. 10 Two moons by K-means

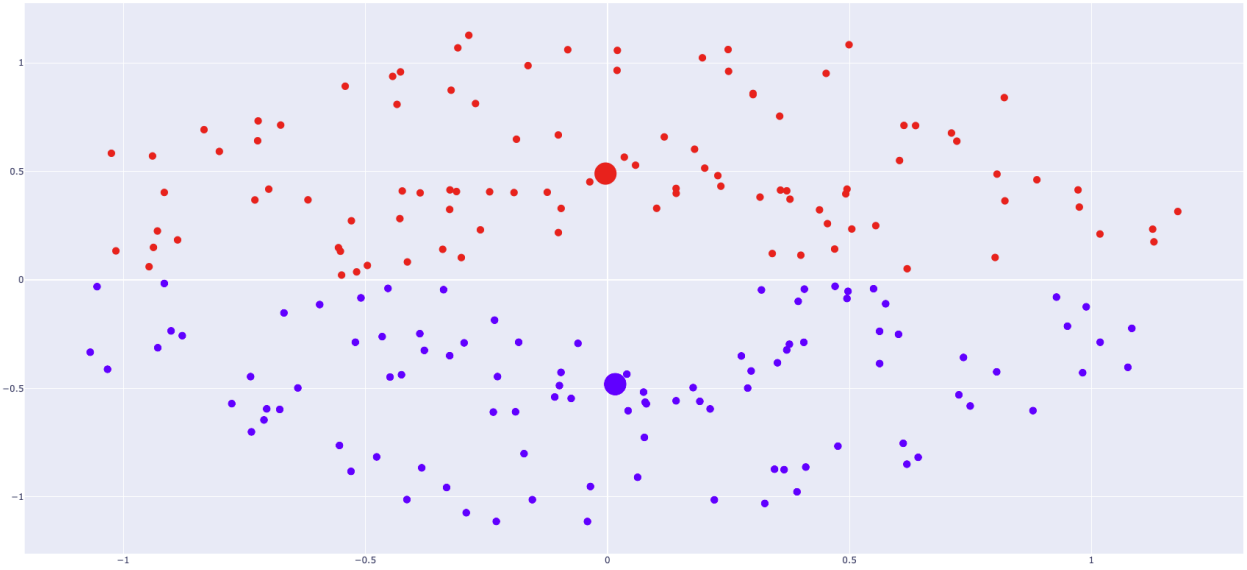

Fig. 11 Two circles by K-means

Comparison

By checking the output of K-means clustering, we found this algorithm didn't work ideally. It generated clusters that are interspersed in both cases. However, Kruskal's algorithm worked better on clustering the same datasets. For example,

from Fig. 7, we could see clearly that the data points are clustered into two moon-shaped groups.
Besides, unlike the K-means algorithm, which needs to iterate multiple times to find the final centroids, Kruskal's clustering is more efficient.

Improvements on the Kruskal's algorithm clustering

Although Kruskal's algorithm works better than K-means on clustering the given datasets, we still found there were some imperfections in this approach. In Particular, we cannot avoid the prominent drawback which is known as the "single link effect". So, we hope to make some improvements to the original approach.

1. Improvements based on distance

First, after reading some related papers, we know that distance metric plays a crucial role in identifying similar data points and forming respective clusters. The original Kruskal clustering method we discussed before uses Euclidean distance as a default distance metric to tell the distance between two points in space, but there are other types of distances that can be used in clustering, such as Chebyshev distance, Manhattan distance, Mahalanobis distance, etc.. Therefore, It is important to play around with other distance measurements for our dataset and see if there are better outputs from using different distance metrics to analyze the clustering performance.

First of all, we need to decipher the differences among those distance metrics. Euclidean distance captures the distance between two data points by aggregating the squared difference in each variable; Manhattan distance captures the same by aggregating the pairwise absolute difference between each variable; Chebyshev distance calculates the maximum of the absolute differences between the features of a pair of data points; Mahalanobis distance

is a multi-dimensional generalization of the idea of measuring how many standard deviations away of a point is from the mean of the distribution. As a result, we can infer from their formulas that if two points are close on most variables, but more discrepant on one of them, unlike Manhattan distance, which will be more influenced by the closeness of the other variables, Euclidean distance will exaggerate that discrepancy and will be more likely to be influenced by outliers. Will Manhattan distance give a more robust result? With this question in mind, we tried to replace the default distance measurement with Manhattan and the output is shown as below:

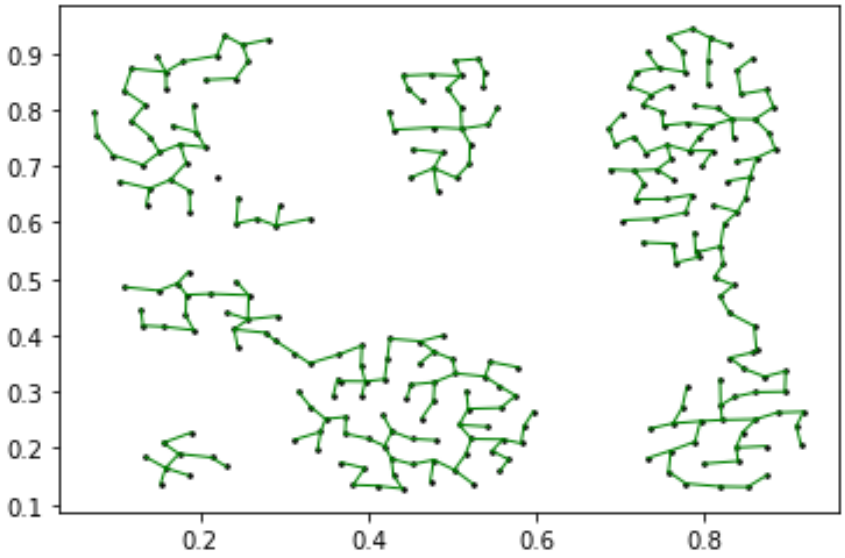

Fig. 12 Single Linkage by Kruskal(Manhattan Distance)

As we can see in the image above, there is no evident improvement on segmenting and we still cannot achieve a desired result.
Similarly, we tried to implement the Kruskal clustering using Chebyshev distance(Fig13). We can see that the single linkage problem has been improved on the right part, but on the left part the problem still exists.

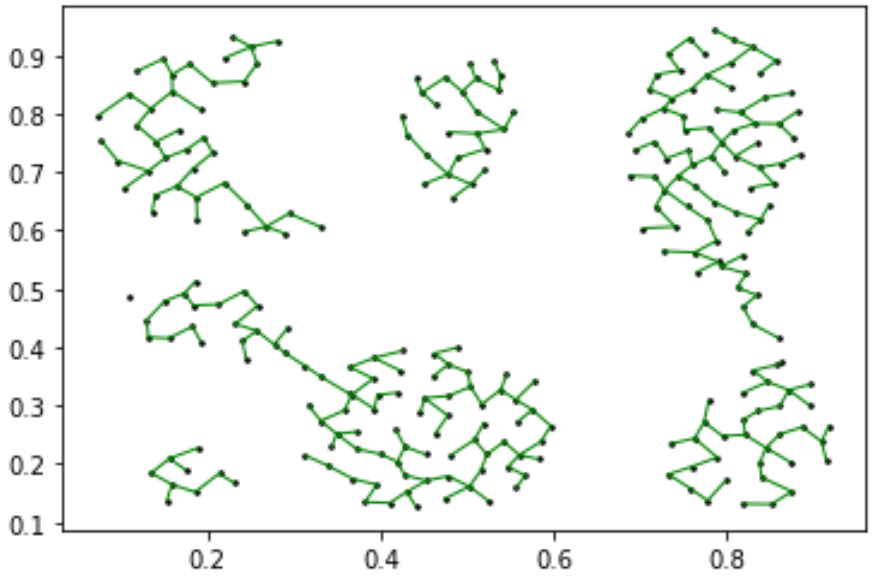

Fig. 13 Single Linkage by Kruskal(Chebyshev Distance)

Finally, we replace the original Euclidean distance with the Mahalanobis distance. The output is as Fig.14.

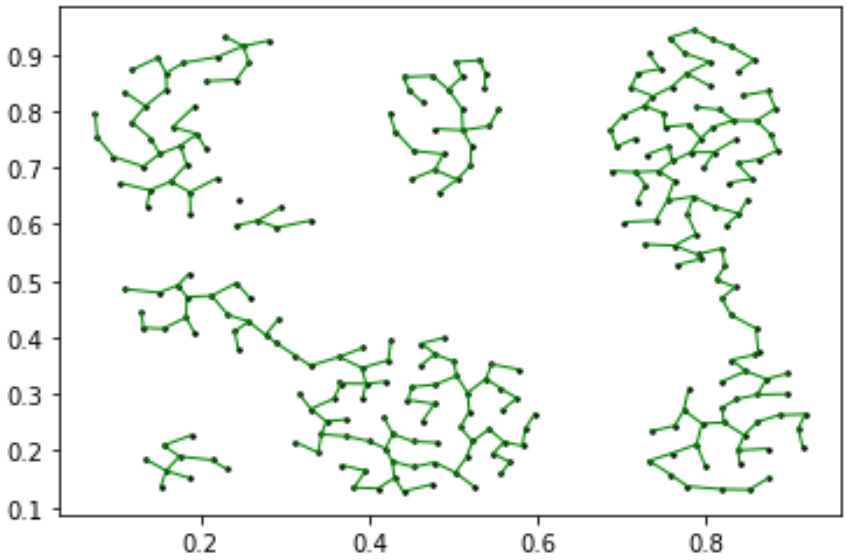

Fig. 14 Single Linkage by Kruskal(Mahalanobis distance)

By comparing with the original Kruskal clustering, we didn’t find improvement on “single link effect” from the original clustering. So, we conclude that the

Mahalanobis distance is not ideal to improve our original clustering algorithm, and one of the reasons may be the shapes of our clusters are irregular. On a two- dimensional basis, the covariance matrix can be seen as some coordinate transformation on the original graph, such as scaling ,shearing and rotating. However, if the original graph has a complex pattern, then the operations would not achieve a good result on every part of the graph.

From the above figures, we could find that the clustering results are not obviously different from the one in our original approach. The main reason might be that single-linkage can be improved locally if we apply some simple geometric transformation corresponding to one part of the graph(as we can see in the right part of Chebyshev result). But it might not be feasible for every part if we have a graph with complex patterns.

2. Improvements based on data point density

The second approach is trying to improve Kruskal Clustering with a density-based clustering algorithm. The main advantages of density-based clustering are 1)its process and corresponding outcome are more comprehensible by human intuition 2)It can handle non-convex shapes in comparison to k-means clustering. Our aim here is to improve Kruskal clustering with some density-based clustering algorithm which might fix “single link effect” issues.

We choose CFSFDP(Clustering by fast search and find of density peaks) as the main approach to improve Kruskal Clustering. In comparison with other density-based algorithms, it can help us determine the density peak positions and the number of clusters.

The finding peak process involves 1)calculating local density 2)finding the nearest distance to one vertex with higher local density for each vertex 3) computing the center scores and sorting.

For small number of samples, we can use Gaussian Kernel to calculate local density:

$$density_i = \sum_{j \neq i} exp(\frac{-d_{ij}^2}{d_c^2})$$

This is one way of calculating local density based on the threshold distance d_c we choose. The number of neighbors is around 1% to 2% of total samples. The neighbor number within d_c would impact calculation results heavily.

Then we have another relative distance to calculate:

$distance_i = min(d_{ij})\ (if\ \exists\ density_j > density_i)$

$distance_i = max(d_{ij})\ (if\ \forall\ j \neq i,\ density_j <= density_i)$

Intuitively, this distance would be small if one vertex has a near neighbor with a higher local density.

A heuristic approach of calculating “center score” is $score_i = density_i \times distance_i$. A vertex with both values high would have a higher chance of being density peak centers.

If we apply this naive approach to pick density peak centers from our “single link graph”. The result is as following:

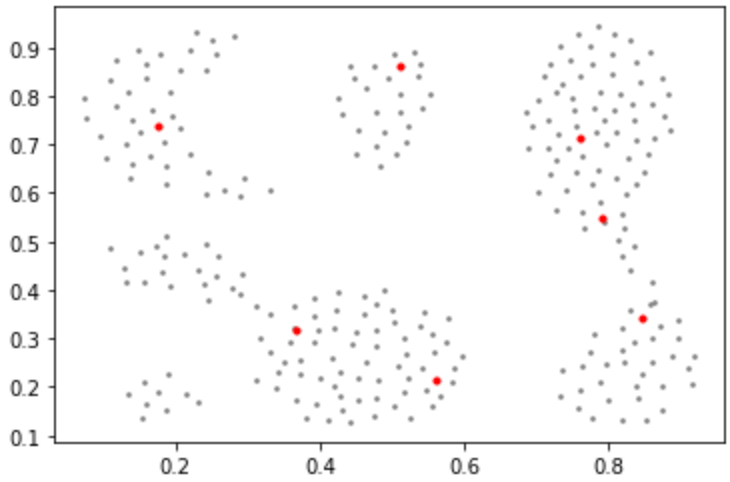

Fig. 15 Naive Density Peak Center Finding (CFSFDP)

We find the center positions are also affected by the “single link effects”. There exist some offsets from where we deem as perfect “peak density” by intuition. CFSFDP scores provide us with numeric metrics for identifying density peaks by picking high scores. Considering the offset of centers, we might also want to take a look at the distribution of “low score” vertices:

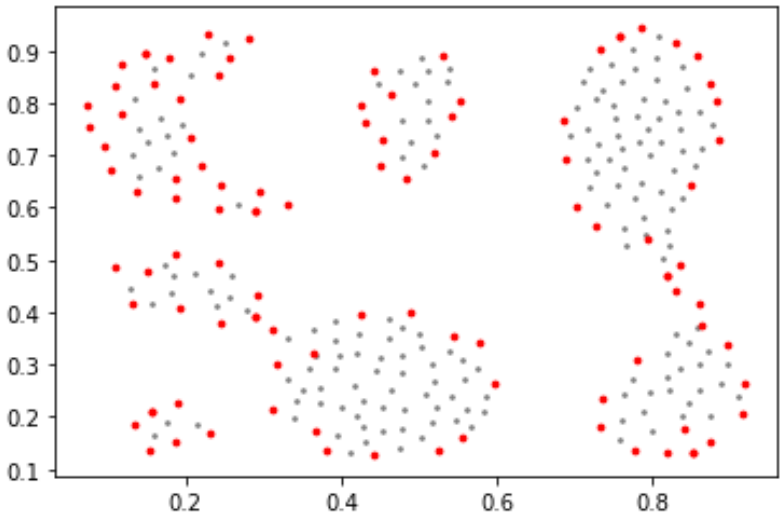

Fig. 16 Naive Low Density Vertices Finding (CFSFDP)

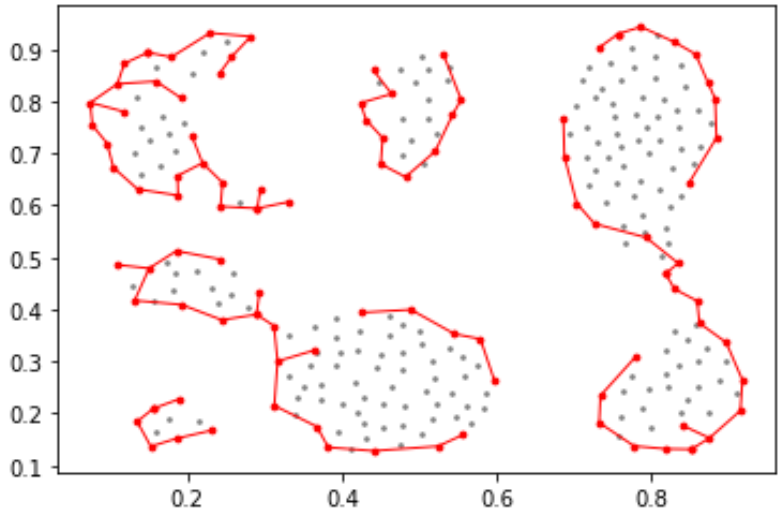

Fig.17 Border and Single Link Detection(CFSFDP)

We can see that the border vertices are classified as low-density vertices and are in accordance with human intuitions. Another finding is that, the “single link” vertices are also classified as low-density vertices.

The above graph gives us some hints about how we can modify CFSFDP to facilitate Kruskal clustering and mitigate “single link effects”. There are several steps:

1. Compute all vertices’ CFSFDP scores
2. Eliminate vertices with low scores(can be 5% to 10%) and use rest vertices to compute CFSFDP scores again. Retrieve peak centers based on fixed scores.

The experiment result is as following(we eliminate about 5% vertices):

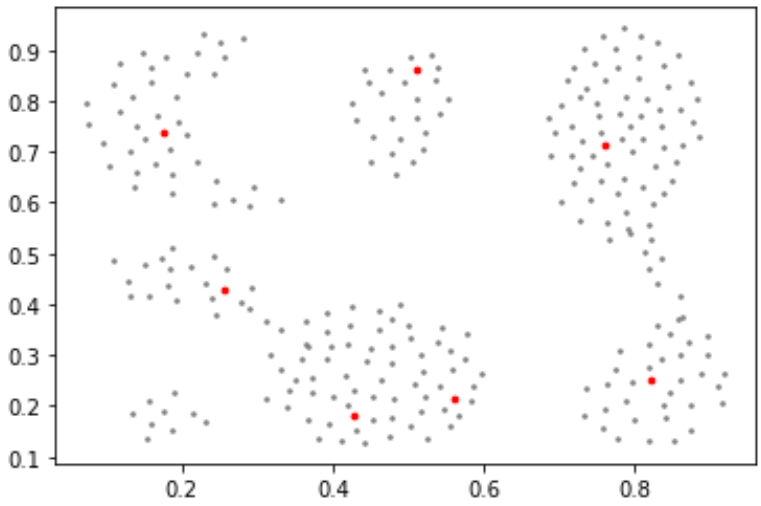

Fig.18 Fixed Density Peak Center Finding(CFSFDP)

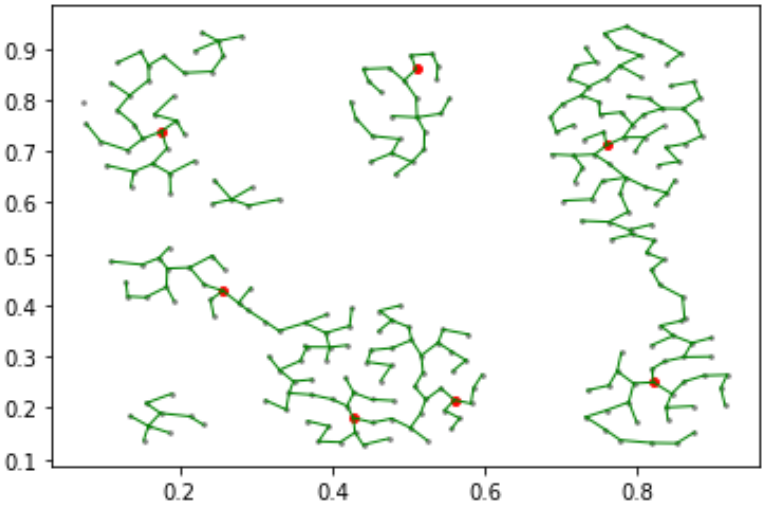

Fig.19 Evaluation of Kruskal Clustering by CFSFDP centers

3. Run Kruskal clustering algorithm to get clustering results.
4. If any cluster contains more than 1 density peak centers from step 2, we would eliminate low density vertices from this cluster first. Then we would use the number of density peak centers as the number of clusters(k > 1) to run Kruskal clustering to divide this cluster again.

The following is the improved clustering results after above steps:

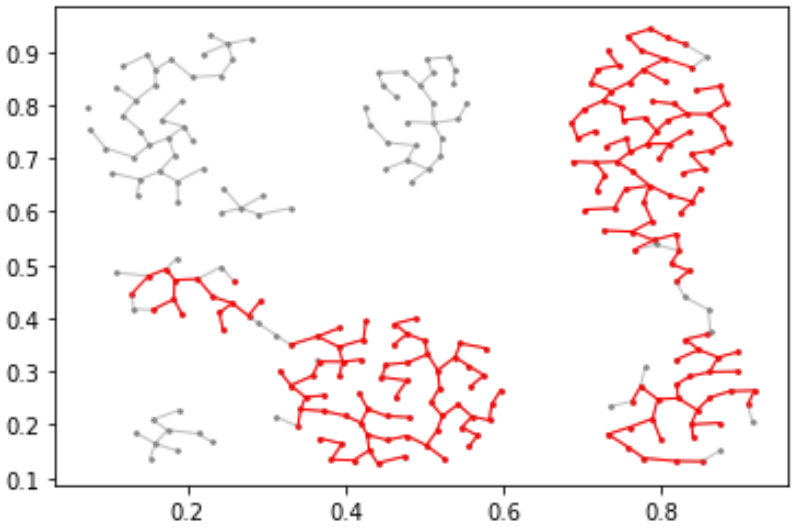

Fig.20 Improved Kruskal Clustering by CFSFDP

The gray parts are original Kruskal clustering results. The red parts are fixed Kruskal clustering results. We can see that the “single link effects” are mitigated. The tradeoff is that we might lose some vertices during the elimination process. There can be one more step to recover the lost vertices using cluster centers. However, the recovery process is out of the scope of Kruskal clustering. Adopting CFSFDP would not increase total time complexity as the time complexity of all the steps would not exceed O(n^2logn). In the naive Kruskal clustering, we would have to sort all the edges and the time complexity is O(n^2logn).

Conclusion

Kruskal clustering has its own advantages and disadvantages. We compare it with k-means clustering in terms of algorithm details, clustering results and time complexity.

We have tried different approaches to mitigate“single link effects”. We find that the density-based clustering approach can help in improving the Kruskal clustering results in comparison with modifying distance calculations.

Yangli Liu: For our research, I noticed that Kruskal's algorithm seems to be a clever way of computing single-linkage clustering. Clustering is a big topic in machine vision that groups similar items into clusters. And single-linkage clustering is based on grouping clusters in a bottom-up fashion, at each step combining two clusters that contain the closest pair of elements not yet in the same cluster into one. Based on the research on K-clustering, we computed the threshold distance of two clusters, namely, the spacing of the clustering. If every pixel of the image has a function of individual RGB value, then the difference between two pixels can be calculated and defined by the spacing of the clustering. I also learned the fact that the K-clustering algorithm surely has its weakness in separating some points if they have a linkage shape of points that join them together. If we only adapt the original Kruskal algorithm using Euclidean distance for clustering, for some edge cases, the computer's result will be contradicted to our human being's intuition. I hope this research will be the first step for me to dig into the big picture of image recognition. Most importantly, it is my endeavor to come up with a better algorithm for this big topic.

Jieqi Yang: By completing this project with my teammates, I got a deeper and better understanding of Kruskal's Algorithm. During the exploration of how Kruskal's Algorithm works in clustering, we also found both the pros and cons of this algorithm which were shown by our clustering outputs. Besides, we also tried some solutions to improve the original clustering approach. It can be said that this project broadens our thinking perspectives. To check the performance of our Kruskal clustering approach, we compared it with the K-means clustering, which is an introductory clustering technique in machine learning. By learning K-means from scratch, we have a chance to peek into the field of machine learning which seems interesting to me. So, in the following semesters, I would like to choose more machine learning courses and hope to join more related projects.

Lu Wang: First of all, this project provides me with a great opportunity to work with a wonderful team. I learned a lot not only from the research I've done on different clustering algorithms, but also from my teammates. Each one of them has their own perspective about this course and the project topic. Regarding this project itself, my knowledge on Kruskal's algorithm is limited to what I've learned in the course modules. After working on this project, I got a much deeper understanding of Kruskal's algorithm and its application in clustering. At the same time, I got to know the theory behind K-means clustering, which is a popular machine learning technique that involves the grouping of different data points. In our project, we presented our step-by-step approach to compare the clustering performance using two algorithms and we played around with other distance measurements to seek improvement on the clustering performance. My experience in this project has equipped me with a thorough understanding about Kruskal's algorithm and piqued my interest in other machine learning algorithms.

Xiaobo Qian: This project provides us with a good opportunity to deepen our understanding of clustering problems. We have a great team and we undertake different responsibilities. The way of improving Kruskal Clustering is more like an open question for us. We have discussed different approaches and tested with different techniques. During the process of exploring various possibilities, we learned the importance of collaboration and how a flexible code framework would help if we have new ideas and code changes. Although there are some tradeoffs, we have finally improved the Kruskal clustering algorithm and mitigate "single link effects". We have learned much knowledge of clustering algorithms and their potential applications from this process.

References

https://medium.com/@mitanshupbhoot/comparative-applications-of-prims-and-kruskal-s-algorithm-in-real-life-scenarios-4aa0f92c7abc

https://sites.google.com/site/dataclusteringalgorithms/clustering-algorithm-applications?authuser=0
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5397966

https://www.researchgate.net/publication/11355740_Clustering_gene_expression_data_using_a_graph-theoretic_approach_An_application_of_minimum_spanning_trees
https://docs.google.com/document/d/1wWkZLMgSnQsENh8NjrpcR-pdkRelwFr6SiuC31H48W4/edit#

https://www.geeksforgeeks.org/k-means-clustering-introduction/
http://datamining.rutgers.edu/publication/internalmeasures.pdf
https://en.wikipedia.org/wiki/Mahalanobis_distance

A. Rodriguez and A. Laio,”Clustering by fast search and find of density peaks,”Science,vol.344,pp.1492-1496,June.2014.

P. K. Jana and A. Naik, "An efficient minimum spanning tree based clustering algorithm," 2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS), 2009, pp. 1-5, doi: 10.1109/ICM2CS.2009.5397966.

Wang Peng, Wang Junyi, "A Clustering Algorithm Based on Find Density Peaks," Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering, pp. 81-85, Beijing, 25-27 June, 2017.

Appendix

Clustering based on Kruskal's Algorithm

```
# CS5800 Final Project
# Kruskal Algorithm for Clustering
# Original from addejans
# Modified by Yangli Liu
import sys
import math
import matplotlib.pyplot as plt

class Node:
    def __init__(self, x, y, p):
        self.x = x
        self.y = y
        self.parent = p
        self.rank = 0

class Edge:
    def __init__(self, u, v, w):
        self.u = u
        self.v = v
        self.weight = w

def euclidean_distance(x1, y1, x2, y2):
    return math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))

def Find(nodes, i):
    if (i != nodes[i].parent) :
        nodes[i].parent = Find(nodes, nodes[i].parent)
    return nodes[i].parent

def Union(nodes, u, v):
    r1 = Find(nodes, u)
    r2 = Find(nodes, v)
    if (r1 != r2):
```

```
        if (nodes[r1].rank > nodes[r2].rank):
            nodes[r2].parent = r1
        else:
            nodes[r1].parent = r2
            if (nodes[r1].rank == nodes[r2].rank):
                nodes[r2].rank += 1

def pick(x, y, l, m, c):
    plt.scatter(x, y, label= l, color= c,
            marker= m, s=30)

def draw(w,t):
    # windows title
    plt.gcf().canvas.manager.set_window_title(w)
    # x-axis label
    plt.xlabel('x - axis')
    # frequency label
    plt.ylabel('y - axis')
    # plot title
    plt.title(t)
    # showing legend
    plt.legend()
    # function to show the plot
    plt.show()

def clustering(x, y, k):
    #initialization
    n = len(x)
    edges = []
    nodes = []

    #initialize nodes with xy-coordinates and index
    for i in range(n):
       nodes.append(Node(x[i], y[i], i))

    pick(x,y,'Dot & Line','.','black')

    #initialize edges with the Euclidean distance between coordinates
    for i in range(n):
        for j in range(i+1, n):
            edges.append(Edge(i, j, euclidean_distance(x[i], y[i], x[j],
y[j])))

    edges = sorted(edges, key=lambda edge: edge.weight)
```

```
    #maintain clusters as a set of connected components of a graph.
    #iteratively combine the clusters containing the two closest items by
adding an edge between them.
    num_edges_added = 0
    for edge in edges:
        if Find(nodes, edge.u) != Find(nodes, edge.v):
            num_edges_added += 1
            Union(nodes, edge.u, edge.v)
            plt.plot([nodes[edge.u].x, nodes[edge.v].x],[nodes[edge.u].y,
nodes[edge.v].y],color='green',linewidth=1)
      #stop when there are k clusters
        if(num_edges_added > n - k):
            plt.plot([nodes[edge.u].x, nodes[edge.v].x],[nodes[edge.u].y,
nodes[edge.v].y],color='red',marker = 'o')
            draw('Final','Clustering')
            return edge.weight
    return -1.0


if __name__ == '__main__':
    with open("twoCircles.txt", "r") as f1:
        data1 = [tuple(i.strip().split(",")) for i in f1.readlines()]
    x = [float(i[0]) for i in data1]
    y = [float(i[1]) for i in data1]
    k1 = 2
    pick(x,y,'Dot','.','blue')
    draw('Circle-Initial','Ploting')
    print("The largest possible value of distance between any two\n\
    points from different subsets is at least {0:.10f}".format(clustering(x, y,
k1)))

    with open("twoMoons.txt", "r") as f2:
        data2 = [tuple(i.strip().split(",")) for i in f2.readlines()]
    a = [float(j[0]) for j in data2]
    b = [float(j[1]) for j in data2]
    k2 = 2
    pick(a,b,'Star','*','blue')
    draw('Moon-Initial','Ploting')
    print("\nThe largest possible value of distance between any two\n\
    points from different subsets is at least {0:.10f}".format(clustering(a, b,
k2)))

    with open("normalClusters.txt", "r") as f3:
        data3 = [tuple(i.strip().split(",")) for i in f3.readlines()]
    c = [float(k[0]) for k in data3]
```

```
    d = [float(k[1]) for k in data3]
    k3 = 7
    pick(c,d,'Dot','*','blue')
    draw('Normal-Initial','Ploting')
    print("\nThe largest possible value of distance between any two\n\
    points from different subsets is at least {0:.10f}".format(clustering(c, d,
k3)))

    with open("singleLinkage.txt", "r") as f4:
        data4 = [tuple(i.strip().split(",")) for i in f4.readlines()]
    e = [float(l[0]) for l in data4]
    f = [float(l[1]) for l in data4]
    k4 = 5
    pick(e,f,'Star','*','blue')
    draw('Linkage-Initial','Plotting')
    print("\nThe largest possible value of distance between any two\n\
    points from different subsets is at least {0:.10f}".format(clustering(e, f,
k4)))
```

K-means clustering

#author: Jieqi Yang

```
import numpy as np
import random
import argparse
from plotly import graph_objects as go


class KMeans:

    def __init__(self, k: int) -> None:
        self.centroids = np.array([0] * k)
        self.k = k

    def _init_centroids(self, X: np.array):

        self.centroids = X[random.sample(range(X.shape[0]), self.k)]

    def fit(self, X: np.array) -> np.array:
        self._init_centroids(X)

        i = 0
        while True:
            labels = self._get_label_indices(X, self.centroids)
            new_centroids = self._assign_new_centroids(X, labels)

            if self._distance(new_centroids, self.centroids).sum() == 0:
                break

            self.centroids = new_centroids

            i += 1


    def _assign_new_centroids(self, X: np.array, labels: np.array) ->
np.array:
        new_centroids = []
        for i in range(self.k):
            cluster_members = X[labels == i]
            new_centroids.append(cluster_members.mean(axis=0))
        return np.array(new_centroids)
```

```
    def _get_label_indices(self, X: np.array, centroids: np.array) ->
np.array:
        label_indices = []
        for x in X:
            distances_x_to_centroids = self._distance(x, centroids)
            label_idx = np.argmin(distances_x_to_centroids)
            label_indices.append(label_idx)
        return np.array(label_indices)


    def _distance(self, X: np.array, Y: np.array) -> np.array:

        #ndim() function return the number of dimensions of an array.
        return np.sqrt((X - Y)**2).sum(Y.ndim - 1)


    def predict(self, X: np.array) -> np.array:
        return self._get_label_indices(X, self.centroids)


def generate_plot(X: np.array, labels: np.array, file_name: str, centroids:
np.array):
    fig = go.Figure()
    colors = ['blue', 'red']
    fig.add_trace(
        go.Scatter(
            x = X[:, 0],
            y = X[:, 1],
            mode='markers',
            marker=dict(color=labels, size=10, colorscale=colors)
        )
    )
    centroids_colors = colors[:len(centroids)]
    fig.add_trace(
        go.Scatter(
            x = centroids[:, 0],
            y = centroids[:, 1],
            mode='markers',
            name="centroid",
            marker=dict(color=centroids_colors, size=30, colorscale=colors)
        )
    )
```

```
    fig.update_layout(title="K means clusters", showlegend=False)
    fig.write_html(file_name)


if __name__ == "__main__":

    parser = argparse.ArgumentParser("Run kmeans to a file and generate
labels")
    parser.add_argument("--file-path", required=True, help="input file
path, should be txt delimited by ','")
    parser.add_argument("--k", required=True, help="Expected number of
clusters")
    args = parser.parse_args()

    data = np.loadtxt(args.file_path, delimiter=',')
    k = int(args.k)
    kmeans = KMeans(k)
    kmeans.fit(data)
    labels = kmeans.predict(data)
    output_plot_name = f"{args.file_path[:-4]}_{k}_clusters.html"
    generate_plot(data, labels, output_plot_name, kmeans.centroids)
```

Improvements to the Kruskal's clustering

```
# -*- coding: utf-8 -*-
"""
Created on Sat Apr 23 14:10:44 2022

@author: admin
"""

# CS5800 Final Project
# Kruskal Algorithm for Clustering
# Modified by Xiaobo Qian
import sys
import math
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy as stats

class Node:
    def __init__(self, x, y, p):
        self.x = x
        self.y = y
        self.parent = p
        self.rank = 0

class Edge:
    def __init__(self, u, v, w):
        self.u = u
        self.v = v
        self.weight = w

def euclidean_distance(x1, y1, x2, y2):
    return math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))

def chebyshev_distance(x1, y1, x2, y2):
    return max(abs(x1 - x2), abs(y1 - y2))

def manhattan_distance(x1, y1, x2, y2):
    return abs(x1 - x2) + abs(y1 - y2)
```

```
def mahalanobis_distance(x1, y1, x2, y2, inv_covmat):
    a = np.zeros((1,2))
    b = np.zeros((1,2))
    a[0][0],a[0][1] = x1, y1
    b[0][0],b[0][1] = x2, y2
    diff = b - a
    left = np.dot(diff, inv_covmat)
    mahal = np.dot(left, diff.T)
    val = mahal[0]
    return val**0.5


def Gaussian_kernel(index, arr, cutoff, choose):
    ret = 0
    for i in range(len(arr)):
        if i == index or not i in choose:
            continue
        x = arr[i][1]
        ret += math.exp(-(x**2/cutoff**2))
    return ret

def Find(nodes, i):
    if i != nodes[i].parent :
        nodes[i].parent = Find(nodes, nodes[i].parent)
    return nodes[i].parent

def Union(nodes, u, v):
    r1 = Find(nodes, u)
    r2 = Find(nodes, v)
    if (r1 != r2):
        if (nodes[r1].rank > nodes[r2].rank):
            nodes[r2].parent = r1
        else:
            nodes[r1].parent = r2
            if (nodes[r1].rank == nodes[r2].rank):
                nodes[r2].rank += 1

def drawPoints(x, y, l, m, c):
    plt.scatter(x, y, color= c,
            marker= m, s=10)

def draw(w,t):
    plt.gcf().canvas.manager.set_window_title(w)
```

```
    plt.xlabel('x - axis')
    plt.ylabel('y - axis')
    plt.title(t)
    plt.legend()
    plt.show()

def clustering(x, y, k):
    n = len(x)
    edges = []
    nodes = []

    # calculate inverse of sample covariance matrix (f) (for
mahalanobis_distance)
    # data = np.zeros((len(x),2))
    # for i in range(len(x)):
    #     data[i][0] = x[i]
    #     data[i][1] = y[i]
    # cov = np.cov(data.T)
    # inv_covmat = np.linalg.inv(cov)

    for i in range(n):
       nodes.append(Node(x[i], y[i], i))

    drawPoints(x,y,'Dot & line','.','grey')

    dists = [[[0 for k in range(2)] for i in range(n)] for j in range(n)]

    #initialize distance matrix
    for i in range(n):
        for j in range(i+1, n):
            val = euclidean_distance(x[i], y[i], x[j], y[j])
            dists[i][j][0] = j
            dists[i][j][1] = val
            dists[j][i][0] = i
            dists[j][i][1] = val

    distsSorted = [[0 for i in range(n)] for j in range(n)]
    for i in range(n):
        distsSorted[i] = list(map(lambda x: x[1], sorted(dists[i],
key=lambda ele: ele[1])))

    # cutoff distance for CFSFDP
    cnt = 4
```

```
    cutoff = sum(map(lambda x:sum(x[:cnt + 1]),distsSorted))/(cnt *
len(distsSorted))

    # compute scores for all vertices
    chooseSet = set()
    for i in range(n):
        chooseSet.add(i)
    denseRec = [0 for i in range(n)]
    for i in range(n):
        denseRec[i] = Gaussian_kernel(i, dists[i], cutoff, chooseSet)

    delta = [0 for i in range(n)]
    for i in range(n):
        minDist = float("inf")
        maxDist = -1
        for j in range(n):
            if i != j and denseRec[j] > denseRec[i] and dists[i][j][1] <
minDist:
                minDist = dists[i][j][1]
            if i != j and dists[i][j][1] > maxDist:
                maxDist = dists[i][j][1]
        if minDist == float("inf"):
            delta[i] = maxDist
        else:
            delta[i] = minDist
    scores = {}
    v = [[0 for i in range(2)] for j in range(n)]
    for i in range(n):
        v[i][0] = i
        v[i][1] = denseRec[i] * delta[i]
        scores[i] =  denseRec[i] * delta[i]


    # pick vertices based on scores
    v_sorted = sorted(v, key=lambda ele: ele[1])
    k2 = math.floor(0.1 * n)
    kleft = n - k2
    chooseSet = set()
    chooseArr = []
    for i in range(k2, n):
        node = nodes[v_sorted[i][0]]
        # drawPoints(node.x,node.y,'Dot & line','.','red')
        # plt.scatter(node.x, node.y, color='red', marker= '.', s=30)
```

```
            chooseSet.add(v_sorted[i][0])
            chooseArr.append(v_sorted[i][0])
        for i in range(0, k2):
            node = nodes[v_sorted[i][0]]
            # drawPoints(node.x,node.y,'Dot & line','.','blue')
            # plt.scatter(node.x, node.y, color='blue', marker= '.', s=30)


        for v in chooseSet:
            nodes[v].parent = v
        chooseEdges = []
        for i in range(len(chooseSet)):
            v1 = chooseArr[i]
            for j in range(i + 1, len(chooseSet)):
                v2 = chooseArr[j]
                chooseEdges.append(Edge(v1, v2, dists[v1][v2][1]))
        # buildMST(chooseEdges, nodes, chooseSet, k, 'red')

        # centers based on new calculated scores on picked vertices
        k3 = 7
        chooseScores = computeScores(dists, cutoff, chooseSet)
        n2 = len(chooseScores)
        centers = set()
        for i in range(n2 - k3,n2):
            index = chooseScores[i][0]
            centers.add(index)
            node = nodes[index]
            plt.scatter(node.x, node.y, color='red', marker= '.', s=60)

        # add edges and recover nodes for building MST
        for i in range(n):
            for j in range(i + 1, n):
                edges.append(Edge(i, j, dists[i][j][1]))
                # x1 = nodes[i].x
                # y1 = nodes[i].y
                # x2 = nodes[j].x
                # y2 = nodes[j].y
                # edges.append(Edge(i, j, mahalanobis_distance(x1, y1, x2,
y2,inv_covmat)))

        cSet = set()
        for i in range(n):
            nodes[i].parent = i
```

```
            cSet.add(i)

        # build clusters based on naive Kruskal clustering
        clusters = buildMST(edges, nodes, cSet, k, 'grey', alpha=0.5)

        # compute the number of centers in each cluster
        centerCnt = {}
        for k in clusters.keys():
            for center in centers:
                cluster = clusters[k]
                if center in cluster:
                    if k not in centerCnt:
                        centerCnt[k] = 1
                    else:
                        centerCnt[k] = centerCnt[k] + 1

        # divide clusters if one cluster contains more than 1 centers
        for k in centerCnt.keys():
            if centerCnt[k] > 1:
                divCnt = centerCnt[k]
                cluster = clusters[k]
                selScores = []
                for index in cluster:
                    selScores.append([index, scores[index]])
                tempScores = sorted(selScores, key=lambda x:x[1])
                n_c = len(cluster) # number of vertices in this cluster
                k4 = math.floor(0.9 * n_c) # size of reserved vertices
                reservedVertices = []
                cNodes = set()
                cEdges = []
                for i in range(n_c - k4, n_c):
                    reservedVertices.append(tempScores[i][0])
                    cNodes.add(tempScores[i][0])
                for v in reservedVertices:
                    node = nodes[v]
                    node.parent = v
                    # plt.scatter(node.x, node.y, color='red', marker= '.',
s=10)
                for i in range(len(reservedVertices)):
                    v1 = reservedVertices[i]
                    for j in range(i + 1, len(reservedVertices)):
                        v2 = reservedVertices[j]
                        cEdges.append(Edge(v1, v2, dists[v1][v2][1]))
```

```
            divClusters = buildMST(cEdges, nodes, cNodes, divCnt, "red")
            del clusters[k]
            for k2 in divClusters:
                clusters[k2] = divClusters[k2]


def computeScores(dists, cutoff, chooseSet):
    denseRec = {}
    delta = {}
    for i in chooseSet:
        denseRec[i] = Gaussian_kernel(i, dists[i], cutoff, chooseSet)
    for i in chooseSet:
        minDist = float("inf")
        maxDist = -1
        for j in chooseSet:
            if i != j and denseRec[j] > denseRec[i] and dists[i][j][1] <
minDist:
                minDist = dists[i][j][1]
            if i != j and dists[i][j][1] > maxDist:
                maxDist = dists[i][j][1]
            if minDist == float("inf"):
                delta[i] = maxDist
            else:
                delta[i] = minDist
    arr = []
    for i in chooseSet:
        arr.append([i, denseRec[i] * delta[i]])
    arr = sorted(arr, key=lambda x: x[1])
    return arr


def buildMST(edges,nodes, cNodes, k, color, alpha = 1):
    clusters = {}
    n = len(cNodes)
    edges = sorted(edges, key=lambda edge: edge.weight)

    #maintain clusters as a set of connected components of a graph.
    #iteratively combine the clusters containing the two closest items by
adding an edge between them.
    num_edges_added = 0
    for edge in edges:
        if Find(nodes, edge.u) != Find(nodes, edge.v):
            num_edges_added += 1
```

```
            Union(nodes, edge.u, edge.v)
            plt.plot([nodes[edge.u].x, nodes[edge.v].x],[nodes[edge.u].y,
nodes[edge.v].y],color=color,linewidth=1, alpha=alpha)
      #stop when there are k clusters
        if(num_edges_added == n - k):
            for i in range(len(nodes)):
                parent = Find(nodes, i)
                if parent not in cNodes:
                    continue
                if parent in clusters:
                    clusters[parent].add(i)
                else:
                    clusters[parent] = set()
                    clusters[parent].add(i)
            return clusters




if __name__ == '__main__':
    with open("singleLinkage2.txt", "r") as f2:
        data2 = [tuple(i.strip().split(",")) for i in f2.readlines()]
    a = [float(j[0]) for j in data2]
    b = [float(j[1]) for j in data2]
    k2 = 7
    drawPoints(a,b,'Star','*','blue')
    draw('Moon-Initial','Ploting')
    clustering(a,b,k2)
```