UCD Michael Smurfit
Graduate Business School

MIS41270 – Data Management and Mining

Assignment 2 – Cluster Analysis

Jesus Cortina Bernal

21200889

Course Name:

MIS41270 – Data Management and Mining

Lecturer:

Dr. Elayne Ruane

Date:

March 25th, 2022

Cluster Analysis

Task 1. K-Means

Datasets Used:

1. Dataset 1: The “Salary.csv” file
2. Dataset 2: 50-point dataset using a numpy rng object with seed of 303.
3. Dataset 3: 30-point dataset using a numpy.rng object with seed of 157.

Seed List for Trials:

The list of seed values for the trials was obtained from random.org (n.d.), with possible seed values from 0 to 1000. The seeds were set using the numpy.rng object (Woodcock, 2021). The 10 seeds were used for both datasets for comparison’s sake. For any other process that required a seed, the seed is specified in the text or in the graph.

Table 1. Random Seeds used during the assignment

Trial	Seed
1	905
2	461
3	875
4	566
5	526
6	812
7	358
8	325
9	299
10	361

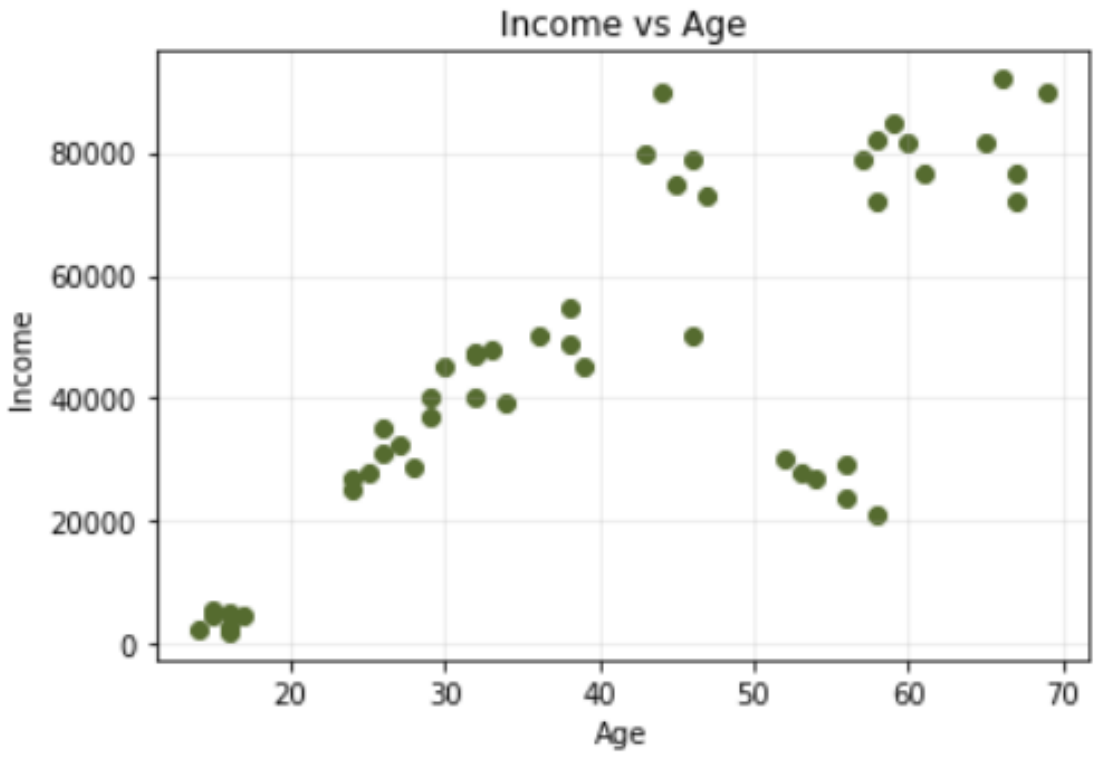
1.1 – K-means in 10 runs

Dataset 1

For the first dataset, I'll start by presenting the scatter plot for the original values without any clusters. Then, I'll present a snippet of the code for one of the trials using the functions from the assignment's Jupyter Notebook (Ruane, 2022), followed by the scatter plots with the results for the 10 trials. Finally, there is a scatter plot for a single run of the algorithm with a n_init value of 10 using the seed 905. For the runs, I decided to go for 5 clusters using the seeds I present at the start of this task's report. I decided to run 5 clusters because at first sight one can kind of visualize 5 agglomerations in the data, and it would be interesting to find if the algorithm discovers that too. The potential clusters I noticed are:

- Less than 20 years and less than $20000
- Between 20 and 40 years with salary between $20000 and $40000
- Between 40 and 50 years with salary over $60000
- Between 50 and 60 years with salary between $20000 and $40000
- Between 60 and 70 years with salary over $60000

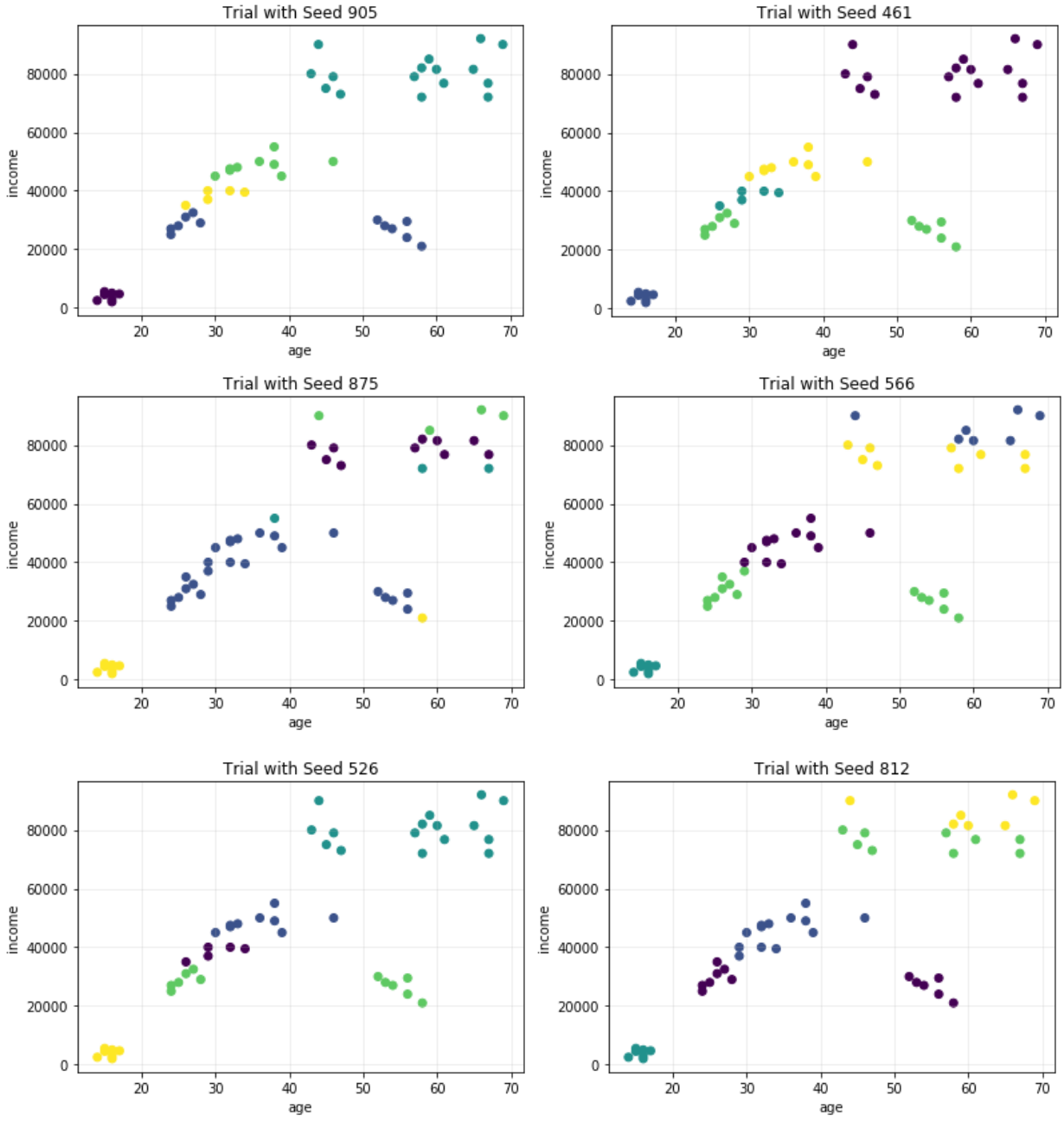
Graph 1 – Scatter Plot of Income vs Age

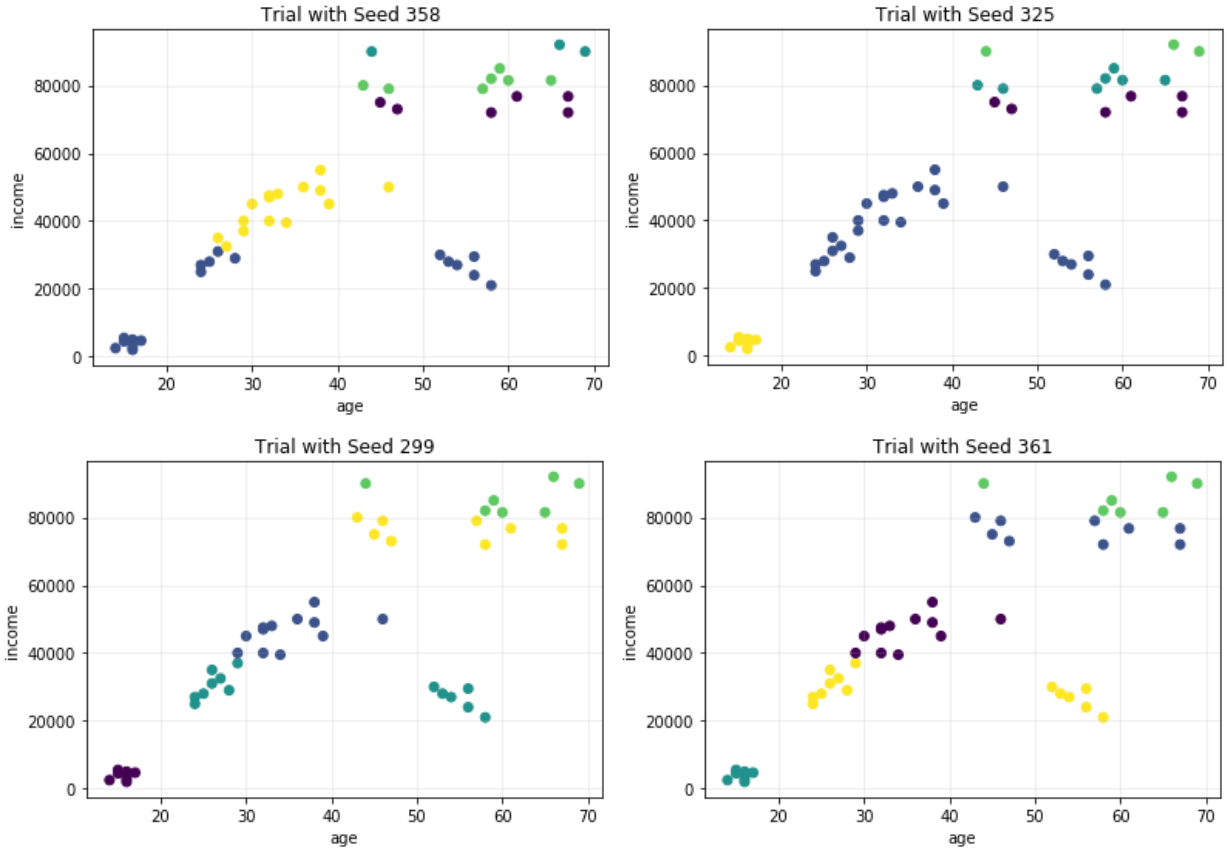


Snippet 1. Cluster Function

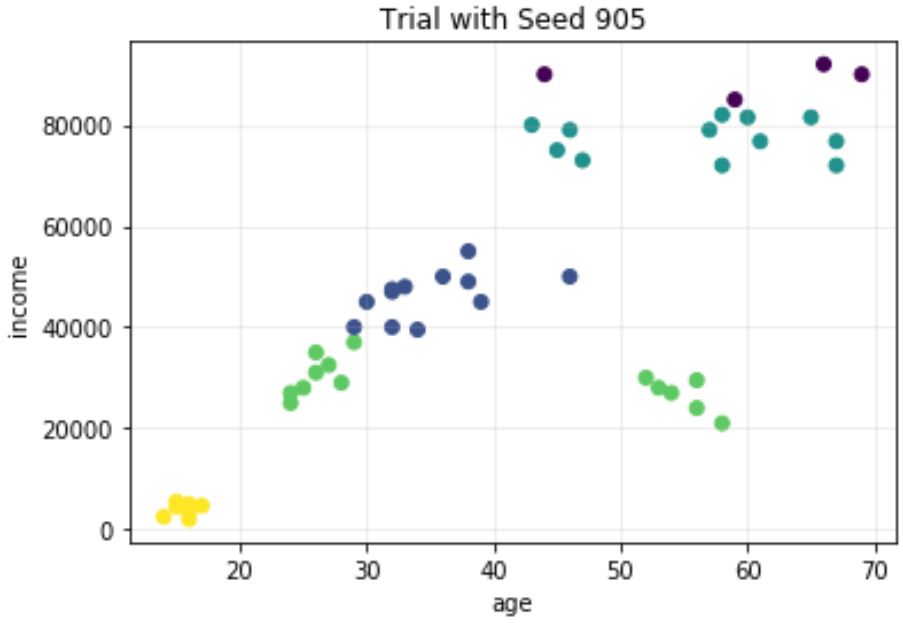
```
run_1_df = run_kmeans(5, salary_data, 905, x_col='age', y_col='income')
```

Graph 2. Scatter Plots for the 10 trials – Dataset 1





Graph 3. Scatter Plot for 10 iterations using seed 905 – Dataset 1



Analysis for Dataset 1

What caught me by surprise for all runs is that the cluster algorithm seems to have prioritized income over age. That is that the clusters seem to be based more around income rather than age. For instance, I'd have expected the points between 50 and 60 years old and income under $40000 to form their own cluster. However, they are being grouped together with the younger people with a similar income rather than people of similar age with higher income. This could mean that the distance between incomes is much more important than the distance between ages in this data set.

As for the clusters itself, the two ones that appear frequently are:

- The cluster of age less than 20 with salary lower than $20000.
- The cluster of people with salaries between $20000 and $40000, regardless of age.
- While not a cluster, the little outlier point around 40-50 age always pairs with the points closer to its income regardless of age.

In contrast, some variations in clustering are:

- Whether the points with incomes between $20000 and $40000 would form one (325, 875), two (299, 812), or even three clusters (526, 461).
- Cluster 358, where the two frequent clusters merged into one
- Whether the points with income above $60000 would form one (461, 526), two (812, 299), or three clusters (325, 358)

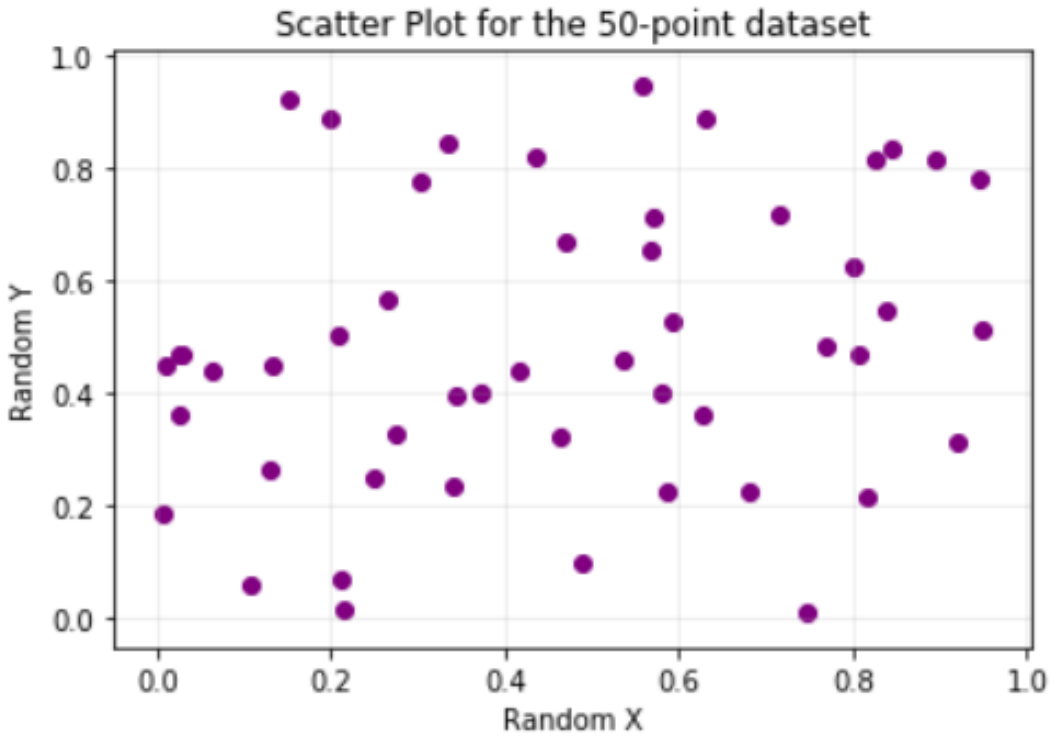
These trends, plus the final cluster with ten iterations at seed 905 lead me to the following conclusions:

- Since the data seems to cluster based on income mostly, the appropriate number of clusters might be 3 or 4 rather than 5. This is because we have the low earners (less than $20000), the high earners (above $60000), and the middle earners who might be partitioned into two clusters or one.
- Still, it is possible for this model to be inaccurate. I suspect this because of the points of people aged 50+ with income of around $30000. Even if visually those points shouldn't combine with those further away, the algorithm put them together. In conclusion, it might be worth complementing these clusters with the results of another algorithm.

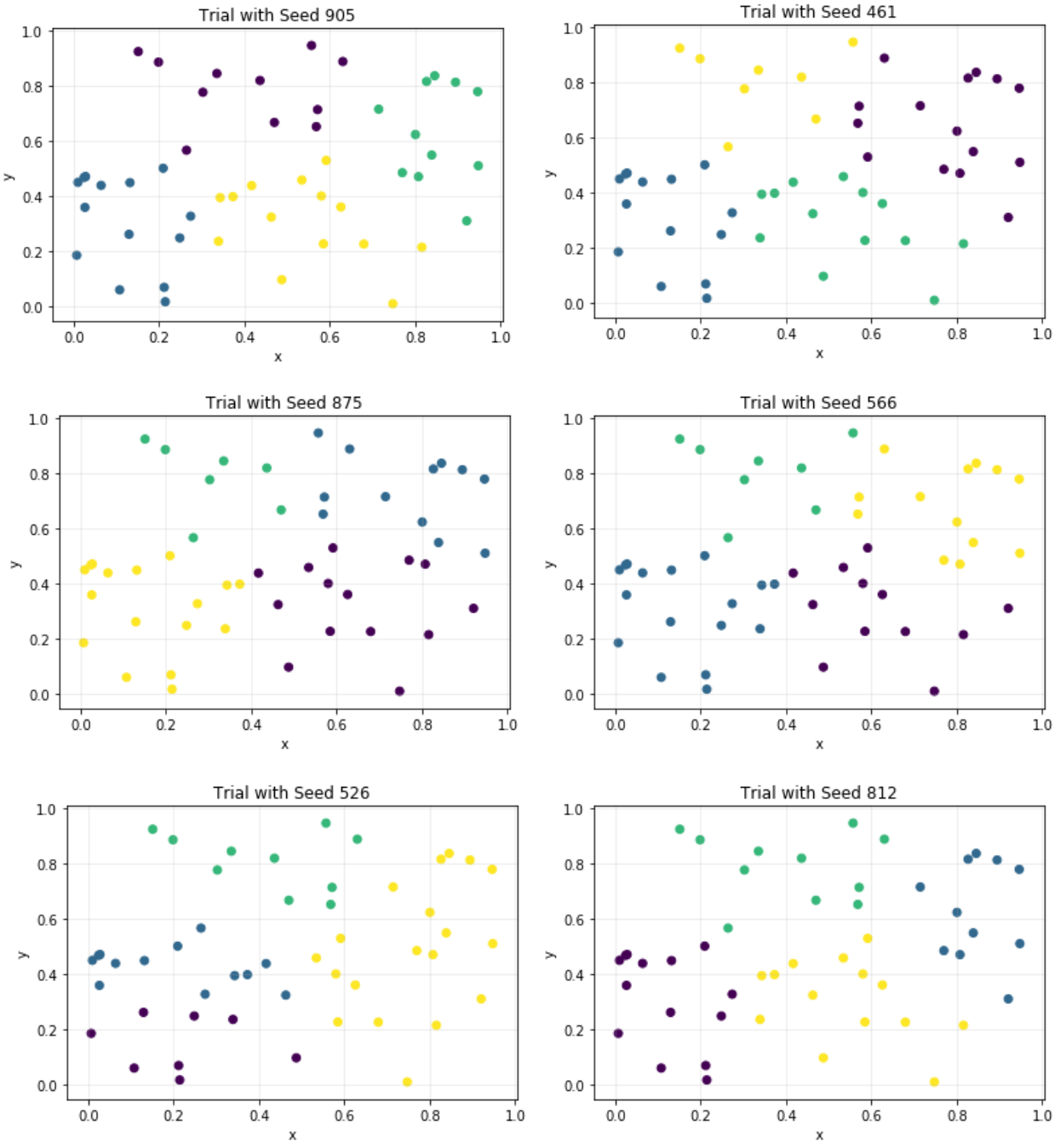
Dataset 2

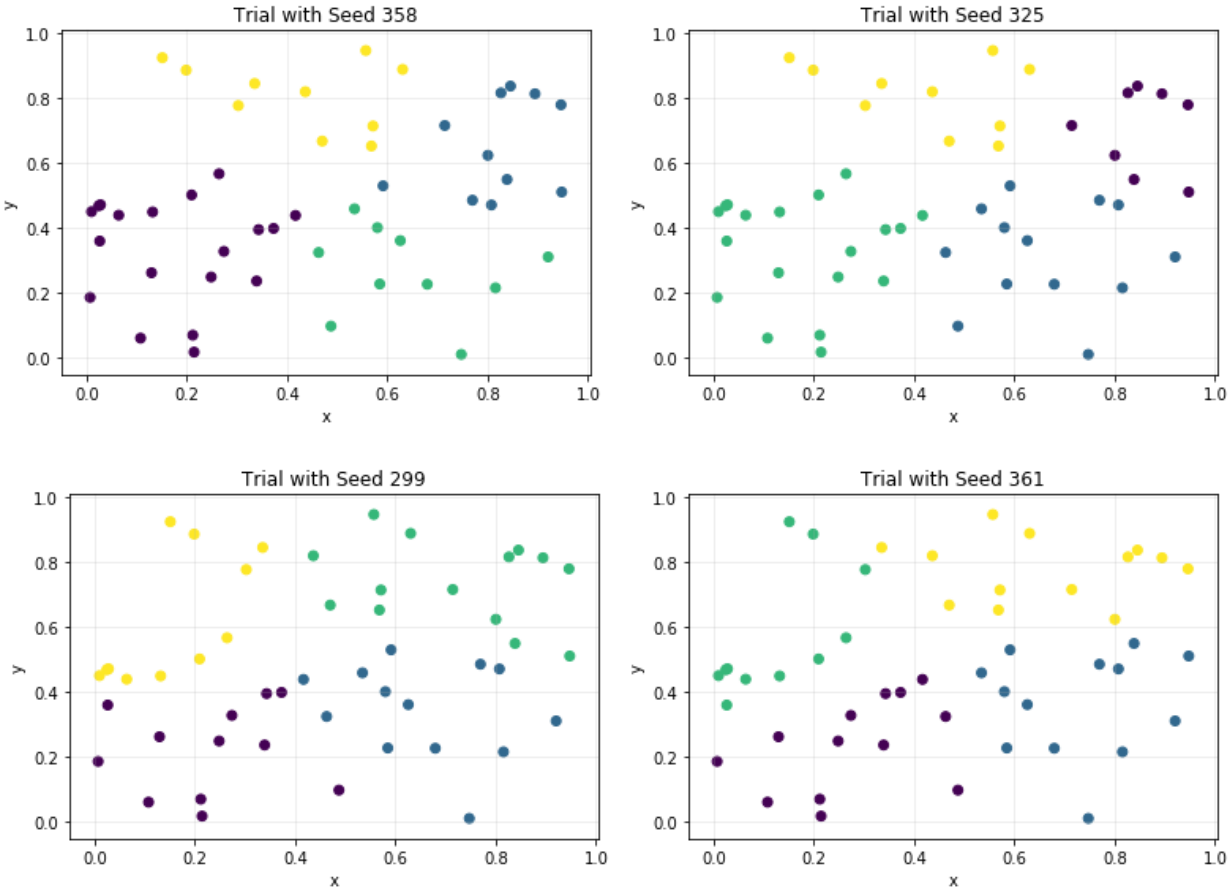
For the second dataset, I also first present the scatter plot for the original values without any clusters. Then, I present the results for the 10 trials. Finally, there is also scatter plot for a single run of the algorithm with a n_init value of 10 using the seed 361. Unlike the previous data set, there are no clear clusters in these points. Thus, I decided to go for a k value of 4 since I consider it might form a proper number of clusters given that I have only 50 points.

Graph 4 – Scatter Plot of Values of X and Y for the 50-point data

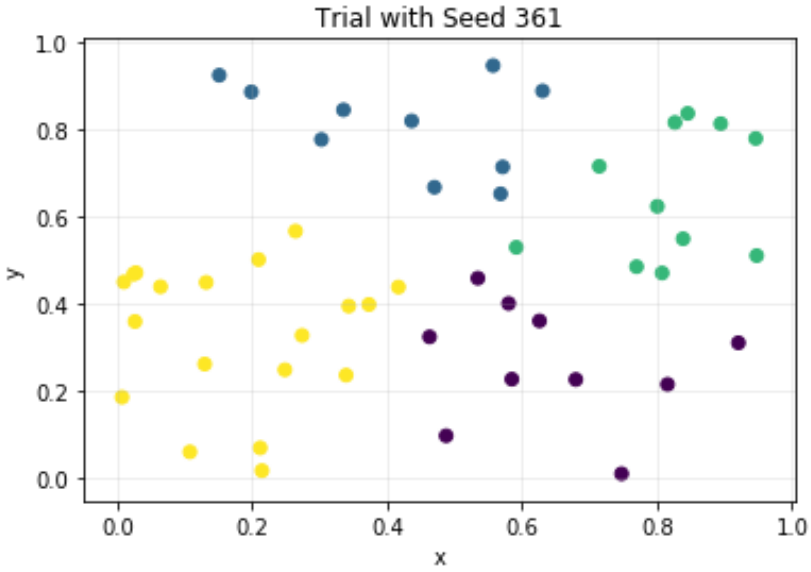


Graph 5. Scatter Plots for the 10 trials – Dataset 2





Graph 6. Scatter Plot for 10 iterations using seed 361 – Dataset 2



Analysis of Dataset 2

Even if the data doesn't appear to have any clear clusters, the runs of k-means found three different cluster forms within the 10 trials. The first form (7 out of 10) involves clusters around the following coordinates:

- Points between x: (0 – 0.4) and y: (0 – 0.5)
- Points between x: (0 – 0.6) and y: (0.5 -1)
- Points between x: (0.4 – 1) and y: (0 – 0.5)
- Points between x: (0.6 - 1) and y: (0.5 – 1)

The second form (Seeds 299 and 361) involves:

- Points between x: (0 – 0.5) and y: (0 – 0.4)
- Points between x: (0 – 0.3) and y: (0.4 -1)
- Points between x: (0.5 – 1) and y: (0 – 0.5)
- Points between x: (0.3 - 1) and y: (0.5 – 1)

The third form (Seed 526) involves:

- Points between x: (0 – 0.5) and y: (0 – 0.3)
- Points between x: (0 – 0.5) and y: (0.3 -0.6)
- Points between x: (0 – 0.6) and y: (0.6 – 1)
- Points between x: (0.5 - 1) and y: (0 – 1)

After comparing all 3 forms, I consider that the most appropriate clusters at k = 4 might come from the first form. First, becomes it is the one that is replicated the most in the 10 single iterations. Even in this run, which started with a seed that produced the second form, the algorithm found that the clusters with the lowest inertia are in the first form. Also, the clusters in this form appear to make sense visually. For instance, the points around the lower values of x and y kind of form an eye in this form. In contrast, the other forms split this eye between two clusters. Similarly, form 3 seems weird since it forms a cluster with all the values on the right. In conclusion, I think that the best clusters here are those in the first form.

1.2 – Describing k-means arguments

n_clusters =

This parameter indicates the number of clusters, and therefore centroids, that we want to end up with, and is often known as k (scikit – learn, n.d.C). Unlike other algorithms, k-means requires the user to input the desired number of clusters before running. This is because it must select a k number of centroids to start running. According to Xu and Tian (2015), this type of algorithm is quite sensible to k. This means that if we set k too low, we end up with clusters that are too big; similarly, if k is too high it results in too many clusters. That's why choosing an appropriate value of k is important, while also comparing the results of multiple runs with different k.

init =

According to the documentation, it serves to specify the initialization method with either "random" for traditional k-means, "k-means++" for k-means ++, or by specifying the starting centroids themselves with a function or an array (scikit – learn, n.d.C). Overall, this parameter specifies how are we going to choose the starting centroid to initialize the algorithm. With "random", the centroids are chosen randomly, while with "k-means++" it chooses based on k-means++. The difference between the two is that by choosing normal k-means, we might end up in local optima (scikit – learn, n.d.B). However, with k-means++, the centroids will be sufficiently separated from one another, leading to better starts and results (scikit - learn, n.d.B). Thus, I think it might be better to begin directly with k-means++. However, the algorithm remains sensible to the amount of k.

n_init =

The final parameter tells us how many times we will reset the algorithm. That is, how many times it will choose different centroids, and therefore different clusters, eventually returning the configuration that minimizes inertia the most (scikit – learn, n.d.C). This must not be confused with the number of iterations of k-means within the algorithm, but rather the number of times the algorithm will run (scikit – learn, n.d.C). By choosing a sufficiently high value of iterations, the program will get multiple final cluster results, and choose the best one for us in terms of inertia. Thought, there is still a risk of the algorithm falling for local optima even with high values of clusters. Still, it is better to run it multiple times rather than just a few.

1.3 – Running with k-means++

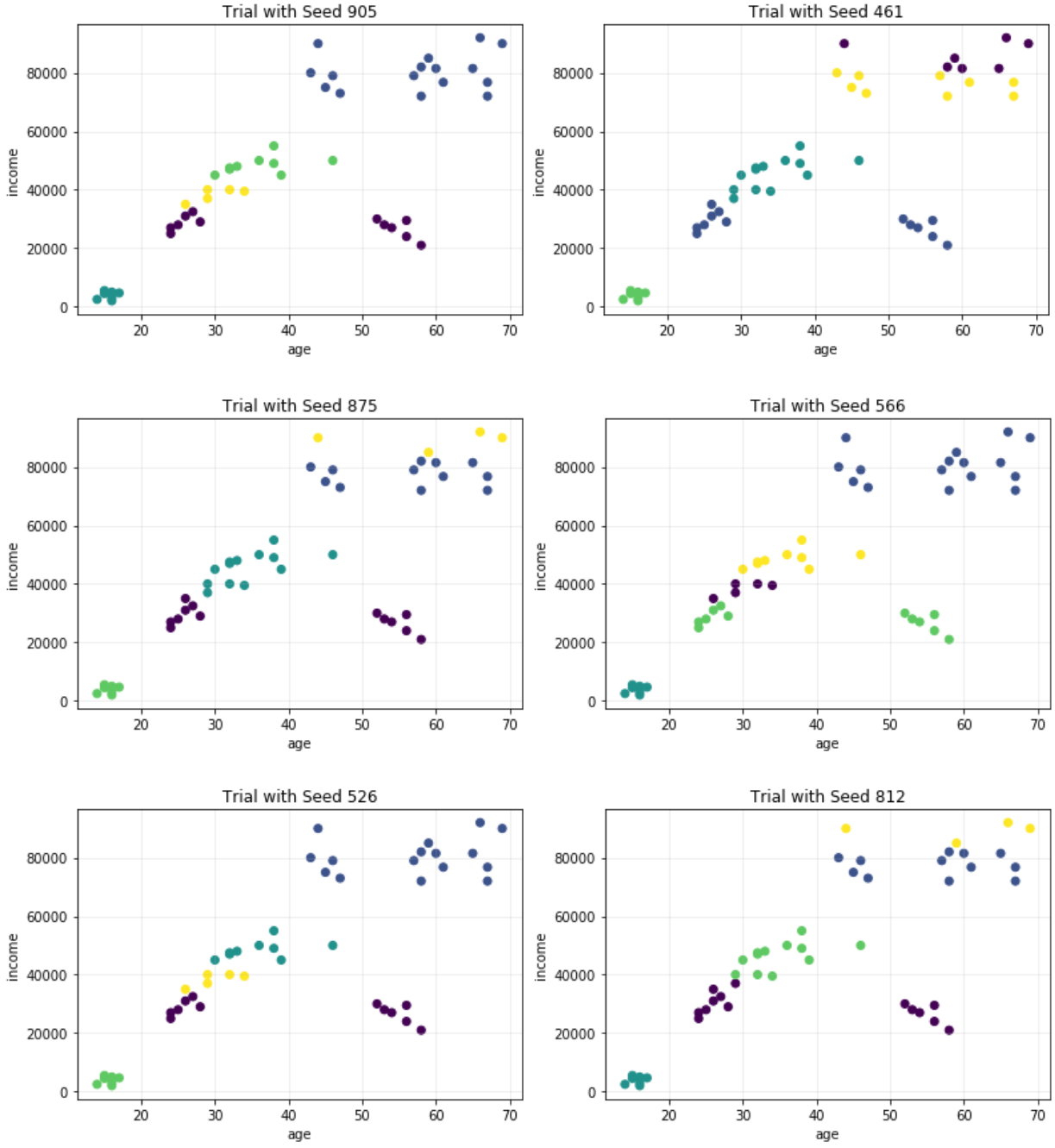
Dataset 1

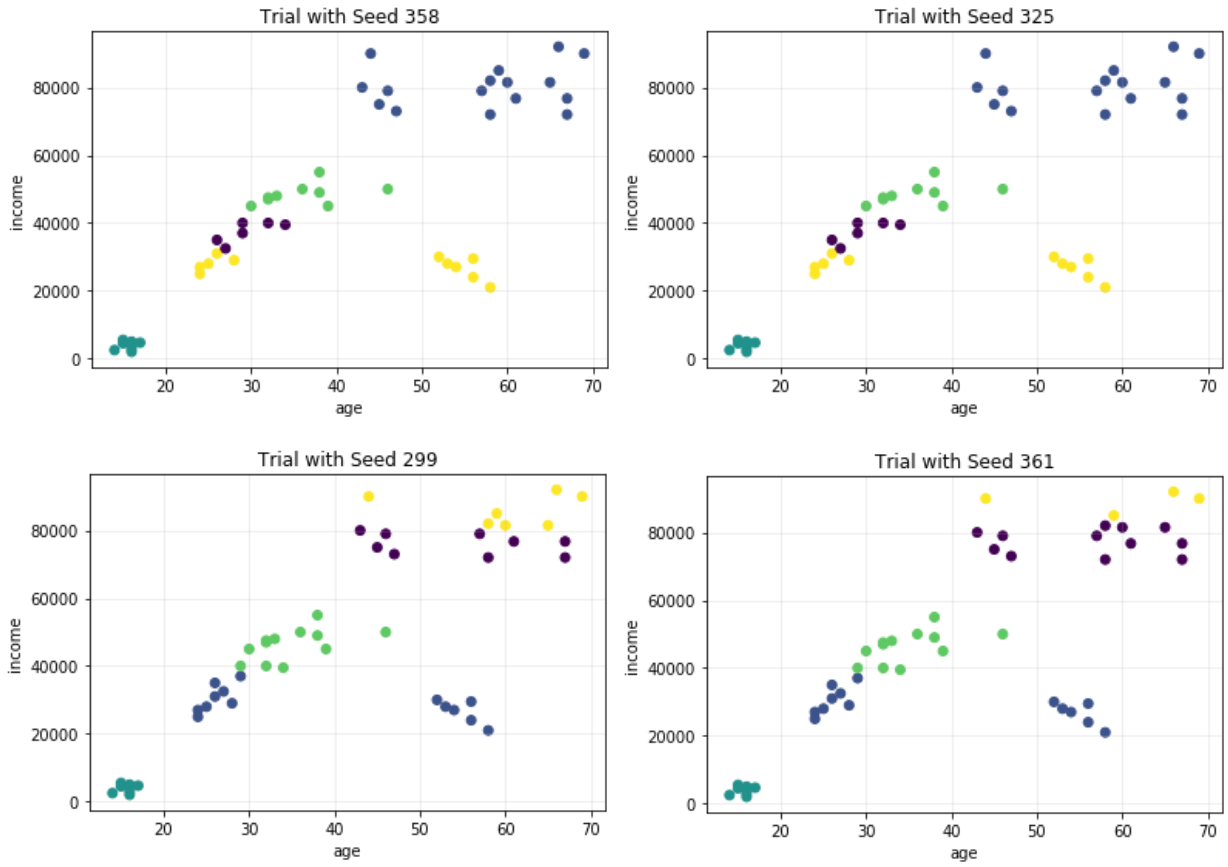
Now, I'll present the results for running 10 runs of k-means++ on the salary dataset. To maintain consistency, I'll use the same 10 seeds and the same number of k as the first runs (5). In addition, I'll also show the snippet of the code for the k-means++ function from the assignment's Jupyter Notebook (Ruane, 2022). Afterward, I'll show the scatter plots with the results of 10 iterations and thereafter the results of a 10-iterations run with 5 clusters. Finally, I'll also show a run of 10 iterations within but with 4 clusters.

Snippet 2. K-means++ function

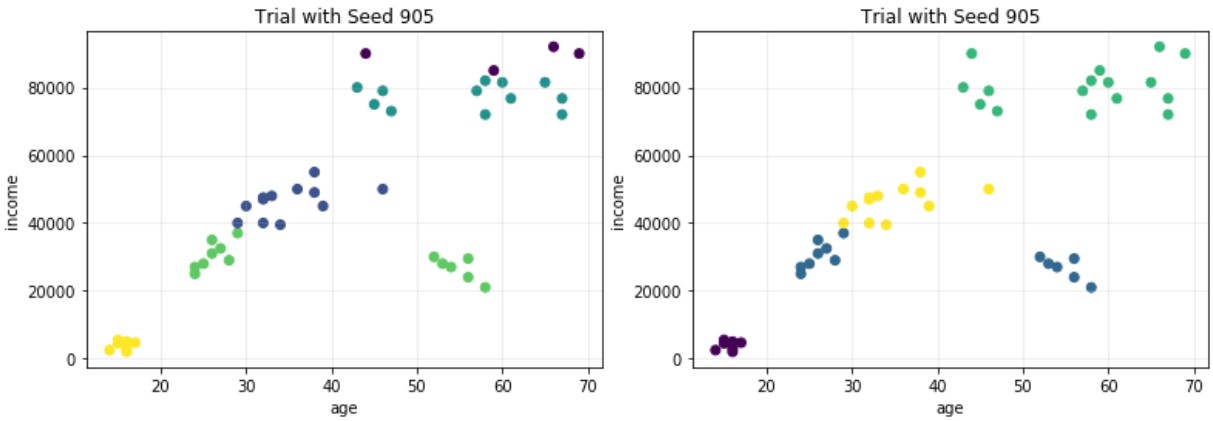
```
def run_kmeans_plus(k, dataset, seed, x_col='', y_col=''):
    # let's make a copy of the dataframe so we can freely add columns
    df = dataset.copy(deep=True) # just for our use case, don't do this as standard


    kmeans = KMeans(n_clusters=k, init='k-means++', n_init=1, random_state = seed)
    kmeans.fit(dataset)
    cluster_labels = kmeans.fit_predict(dataset)
    df[f'cluster_labels'] = cluster_labels
    plt.scatter(df[x_col], df[y_col], c=kmeans.labels_.astype(float))
    plt.xlabel(x_col)
    plt.ylabel(y_col)
    plt.title("Trial with Seed {}".format(seed))
    plt.grid(True, alpha = 0.25)


    plt.show()


    return df
```

Graph 7. Scatter Plots for the 10 trials with k-means++ - Dataset 1





Graph 8. Scatter Plots for 10 iterations of k-means++ using seed 905, with 5 and 4 clusters – Dataset 1



Analysis of Dataset 1 with k-means++

Compared to the results of normal k-means, k-means++ seems to produce some more consistent clusters across all iterations. First, the two clusters that remain mostly stable are:

- The cluster of age less than 20 with salary lower than $20000.
- The cluster of people with salaries between $20000 and $40000, regardless of age.

Then, there are two things happening with the rest of points:

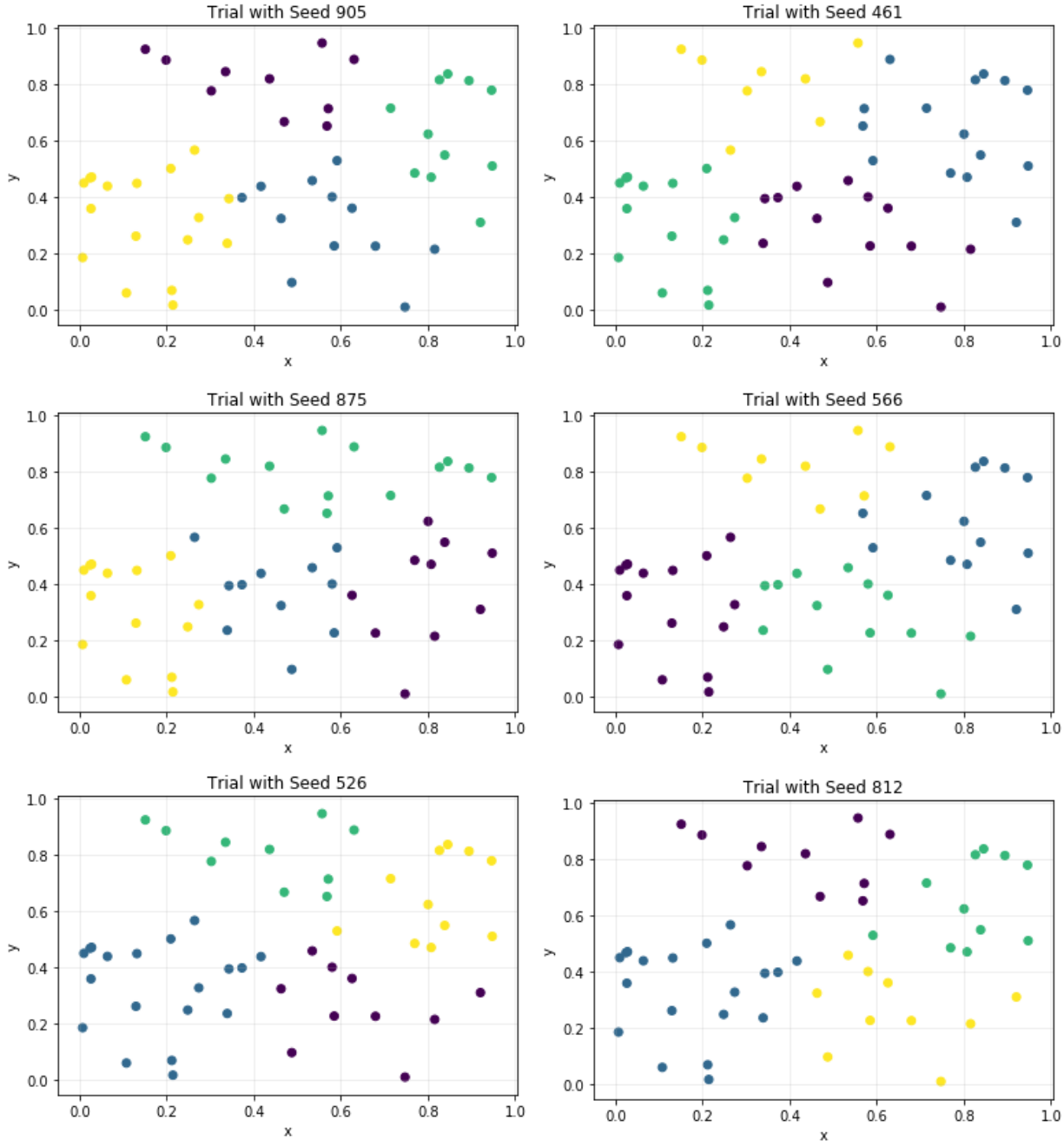
- The points between $40000 and $60000 get split into two clusters or merged into one.
- The points above $60000 also form one or two clusters.

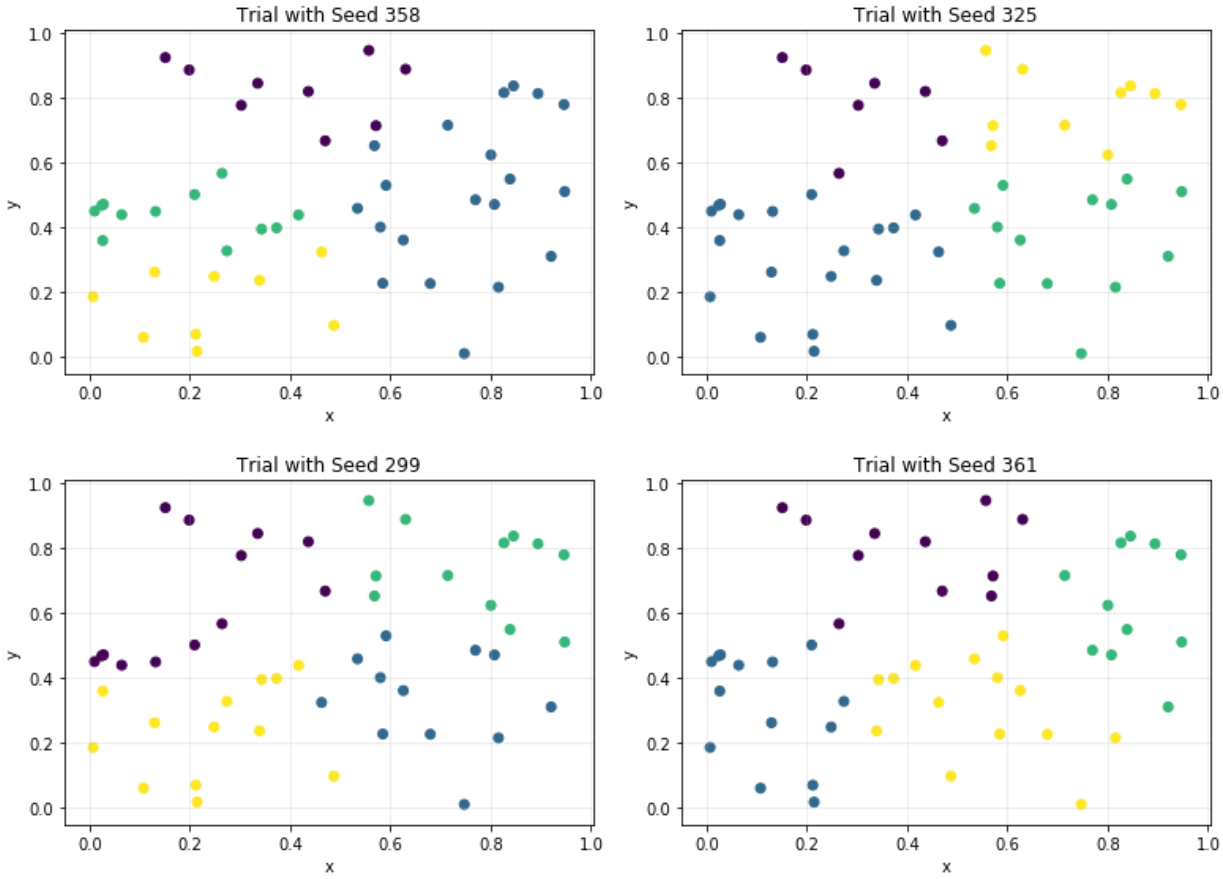
Based on these patterns, the final runs with 10 iterations, plus the findings from the first 10 iterations, I consider that there are 4 main clusters in this dataset based around income rather than age: below $20000, between $20000 and $40000, between $40000 and $60000, and above $60000. If these data were to be used in real life, I'd suggest these 4 groups. Still, I'd add that two of the clusters ($60000+ and $40000 to $60000) might have some subgroups inside which might differ in some small attributes. Finally, I'd still be careful with the cluster between $20000 and $40000. While the algorithm says that those people are quite similar regardless of age, reality might be different. After all, a young man beginning his career is not the same that an old man working at the convenience store. Those two groups are similar, but they might be different in key areas like career prospects or health needs.

Dataset 2

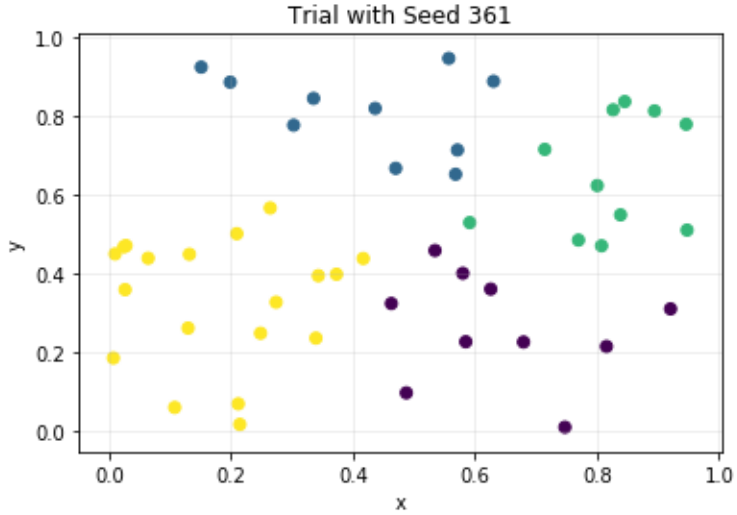
Now, I'll run k-means++ on the second dataset. I'll use the same parameters as the runs with traditional k-means to facilitate comparisons (k = 5, and the same seeds). I'll also make one run with the algorithm at n_init = 10.

Graph 9. Scatter Plots for the 10 trials with k-means++ - Dataset 2





Graph 10. Scatter Plots for 10 iterations of k-means++ using seed 361– Dataset 2



Analysis of Dataset 2

Referring to my findings on the first 10 runs with normal k-means, using k-means++ I also found the first and second clusters forms. There is also a slight variation of the third cluster pattern at seed 358, now with the cluster on the right taking a smaller range of x. In any case, after running the version with 10 iterations the algorithm eventually found out that the best run considering 10 iterations was form 1. For reference, form 1 consists of roughly the next clusters, with a small change in the y-range for the first and second on the list:

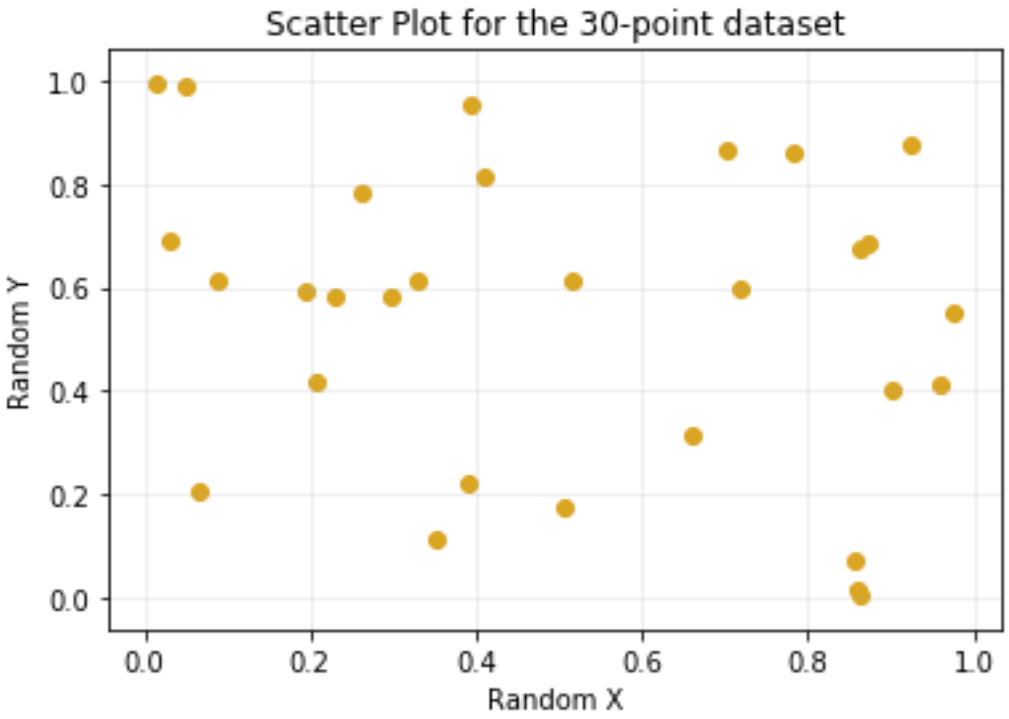
- Points between x: (0 – 0.4) and y: (0 – 0.6) (was 0-0.5)
- Points between x: (0 – 0.6) and y: (0.6 -1) (was 0.5-1)
- Points between x: (0.4 – 1) and y: (0 – 0.5)
- Points between x: (0.6 - 1) and y: (0.5 – 1)

Other thing worth mentioning is that this cluster also appeared exactly the same in some of the 10 runs with k-means++, such as seeds 526 and 812. Overall, based on all the runs for both methods of k-means++ I'd suggest that the best clusters are the ones in form 1. These clusters appear consistently as the best in both methods, and in multiple of the solo runs of the algorithm.

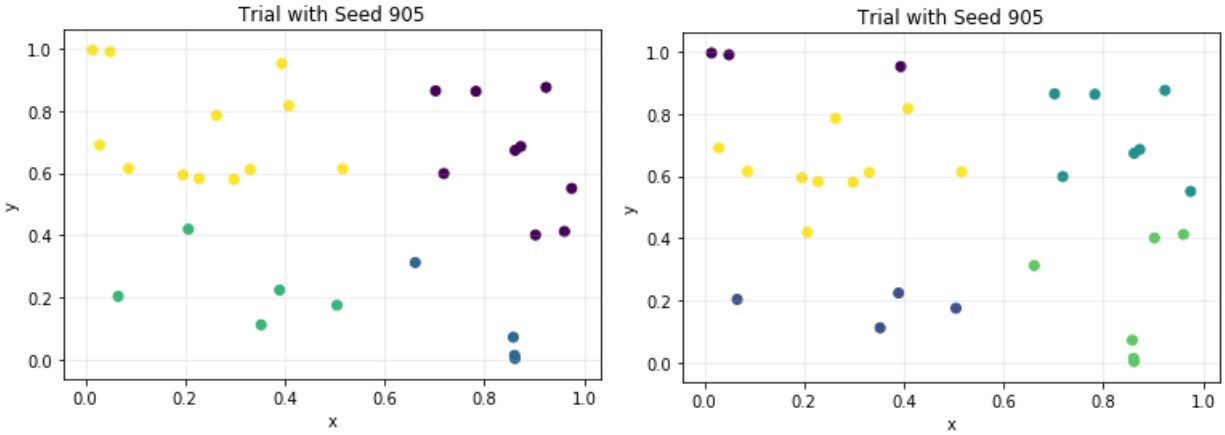
1.4 – Elbow Plot

In this section, I have a created a dataset with 30-points using a seed of 157. Next, I'll show the scatter plot of the original data, then the results of k-means++ with 10 iterations within and k = 4 and 5, and the elbow plot for this dataset. Following that, I'll analyze the results of those charts.

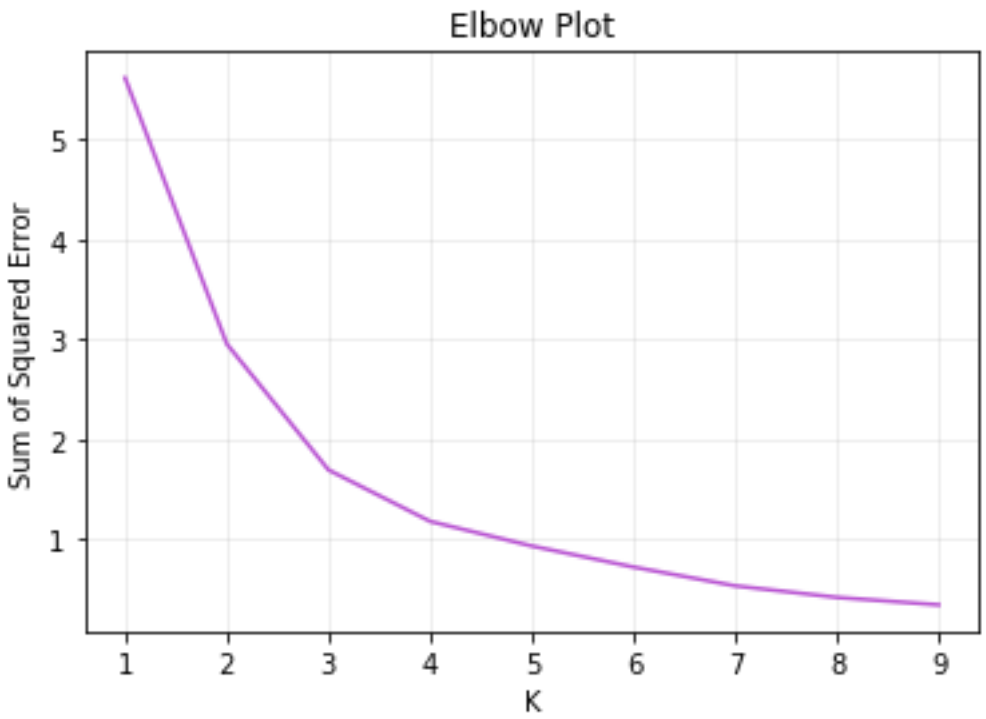
Graph 11. Scatter Plot for the 30-point dataset



Graph 12. Scatter Plots for 10 iterations of k-means++ using seed 905, with 5 and 4 clusters – Dataset 3



Graph 13. Elbow Plot for the 30-point dataset



Analysis of Dataset 3

First, I chose this dataset because I can identify two clusters visually. The first one is around the point at (0.25, 0.8), and the second one around the points near (0.85, 0.65). After running the algorithm, it seems that my intuition was correct. At 4 clusters, those two sections formed a cluster each, while the points below formed another two smaller clusters.

Then, at 5 clusters something interesting happened. While the cluster at (0.85,0.65) remained kind of stable, the other one started to get split for some of the points near the top of it, forming the fifth cluster. While the algorithm signals that this 3-point cluster is significant, I wonder if it really is. I mean, it is a cluster with 3-points, and one of those could easily fit into another cluster.

This point brings me to the elbow plot. This plot signals that after 5 clusters we start minimizing inertia below 1. However, we must consider that having inertia below 1 does not equal having the proper number of clusters. It might happen that some of the points might be too distant from each other, and thus inertia can only be minimized so much. In addition, adding too many clusters can be detrimental to the analysis by generating clusters that are mostly irrelevant even if inertia is lower.

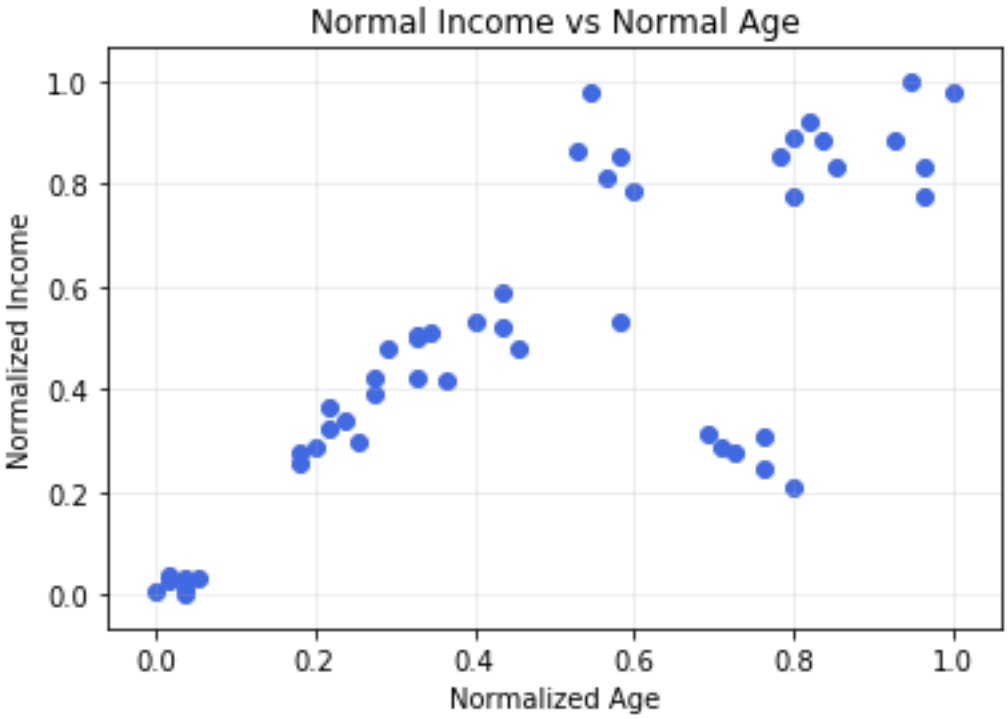
Regarding my dataset, there are two apparent clusters. However, the rest of the points are a bit dispersed. This means that while it might be easy to get low distance values for some points, others are going to have relatively high distance scores. That is why based on the clusters that I obtained with k-values of 4 and 5, I consider that 4 is the most appropriate cluster number. While 5 does have a lower inertia, it generated a cluster that seems irrelevant: the one with the 3 points. Even

then, the inertia value at 4 is not that bad compared to lower values of k. To close, while I'd recommend 4 clusters based on the previous information, there is the possibility that 5 might be the best. While k-means++ offers more reliability than the standard method, there is still some degree of randomness. The seed I used might not have produced the best 5-clusters model, even within 10 iterations. That's why should always try different values of k, but also different runs not to get the exact clusters, but the most correct ones based on the data.

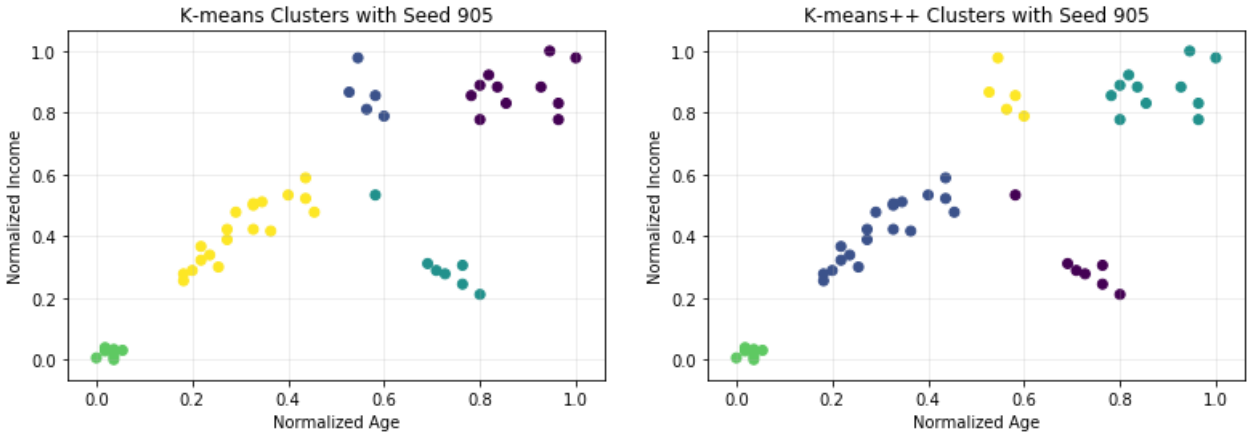
1.5 – Normalizing salary data

To close this section, now I'll analyze the results of the clustering algorithm. First, I'll show the scatter plot with the normalized data, followed by the results of running both normal k-means and k-means++ with a seed of 905. Once again, the value of k chosen is 5 for comparison's sake.

Graph 14. Scatter plot of Normalized Income vs Normalized Age



Graph 15. Scatter Plots for both k-means and k-means++ with seed of 905



Analysis of Dataset 1 – Normalized

The first things that comes out of normalizing the data is that for both methods the 5 clusters I expected at the beginning of all finally appeared. This leads me to believe two things. First, that I was right with my intuition that the distance between salaries matters more than the distance between age for the algorithm. Second, that k-means is quite sensible to magnitudes. The effect of magnitudes is so big that the outlier among the 4 clusters, which originally was being grouped with the cluster to the left, is now being grouped with a cluster on the lower-right corner. That's how big is the difference between normalizing and not normalizing the data.

As my final thoughts for this task, I now realize how having the proper input can alter the results. For example, if an institution would have kept the clusters from the raw data, they would have applied the same politics for people of same income, but different age. That also leads me to another conclusion: that all these algorithms are machines in the end, and that they can only understand so much about the data and its context. That's why human analysis is important. The model might run 1000 times, but it might never detect a cluster that human intuition can see, and it's the analyst's job to discover, handle and understand those gaps and limitations.

Task 2 – Comparing Algorithms

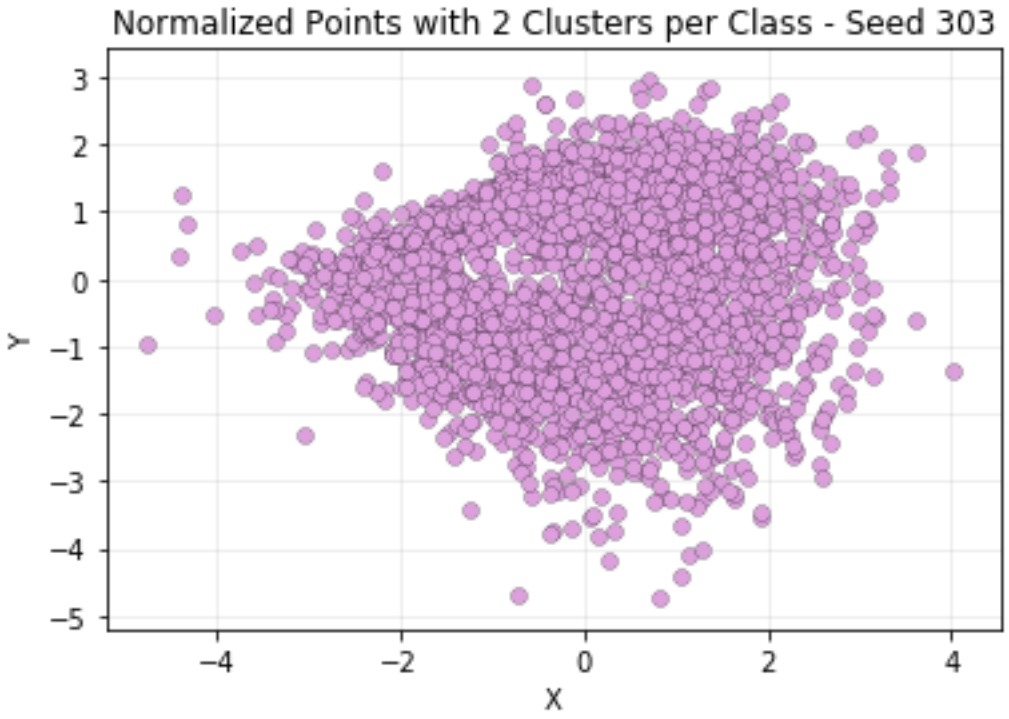
Dataset Description

For this task, I'll be comparing the following cluster algorithms using the same dataset: k-means++, DBSCAN, and Fuzzy c-means. The 3000-points dataset used was generated using the make_classification function from scikit-learn (n.d.D). The parameters used were 2 classes, 2 clusters per class, and a random state of 303. The snippet of the code used to generate this dataset, plus its respective scatter plot, can be found below. As for the algorithms, the number of runs will vary for each of them depending on their parameters. The implementation for each of the algorithms can be found in the Jupyter Notebook file called "21200889_assignment_2_task2".

Code Snippet 3. Creating the 3000-points dataset

```
data_values, class_labels = make_classification(n_samples=3000, n_features=2,
n_informative=2, n_redundant=0, n_classes = 2, n_clusters_per_class=2, random_state= 303)
```

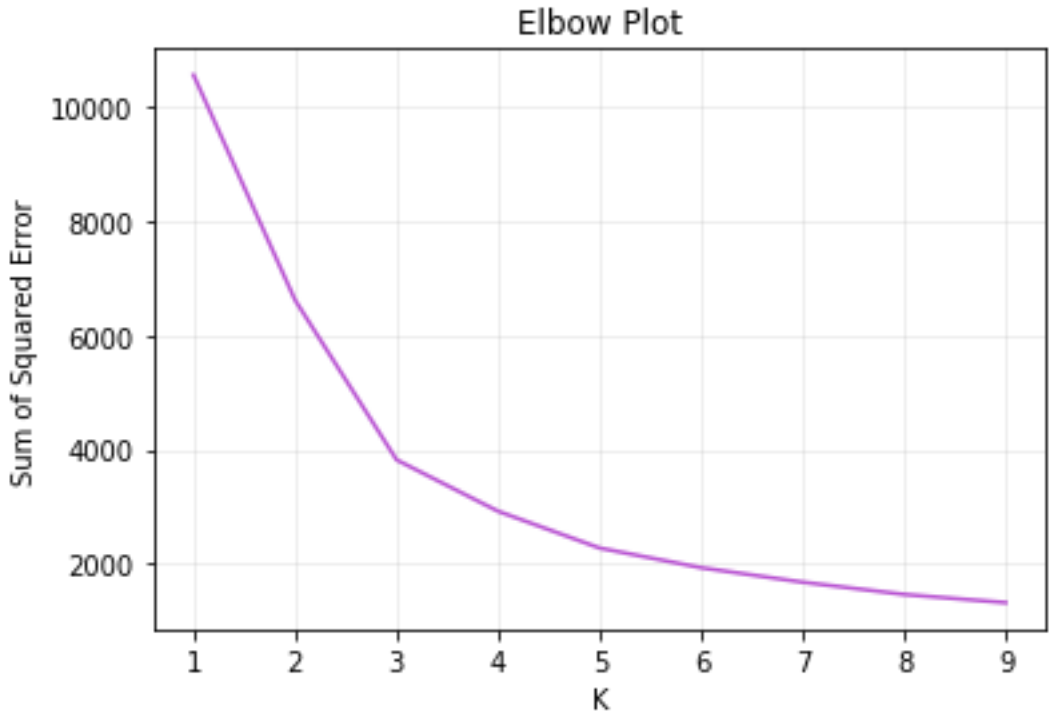
Graph 16 – Scatter Plot with seed 303



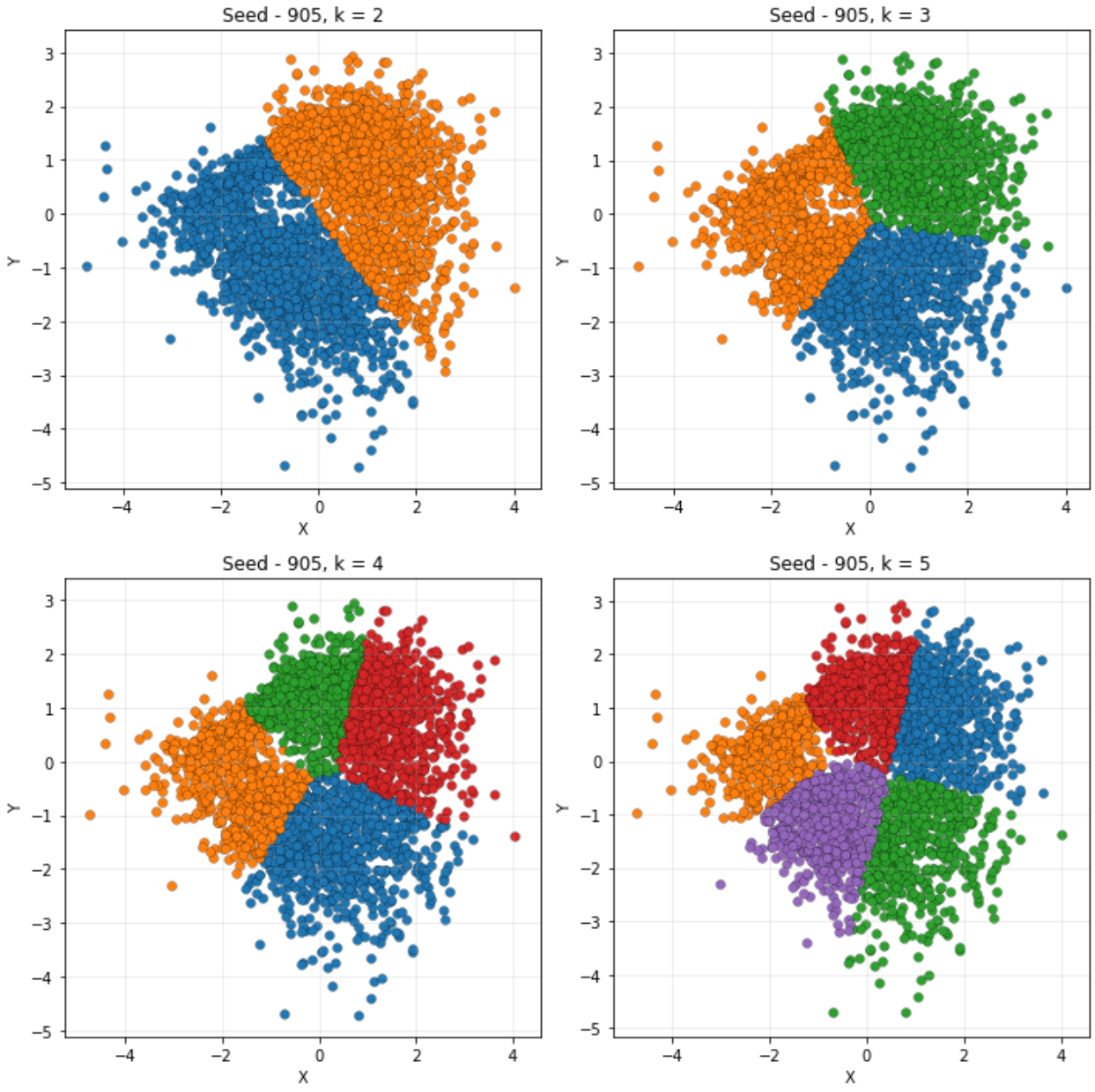
2.1 k-means++

For the run with k-means++, I first decided to draw the elbow plot. Based on this graph, I decided to do the algorithm with values of k from 2 to 5. I decided to use a range instead of a single value of k because I considered that showing different values of k would help me appreciate the benefits of one degree of k versus another. Pus, it would help me identify the best k value. After choosing the values of k, I ran the algorithm using a seed of 905 with 10 iterations per lap. Below, both the elbow plot and the runs of k-means++ can be found.

Graph 17 – Elbow Plot for the 3000-points dataset



Graph 18 – Clusters with k-means++ with respective k-values



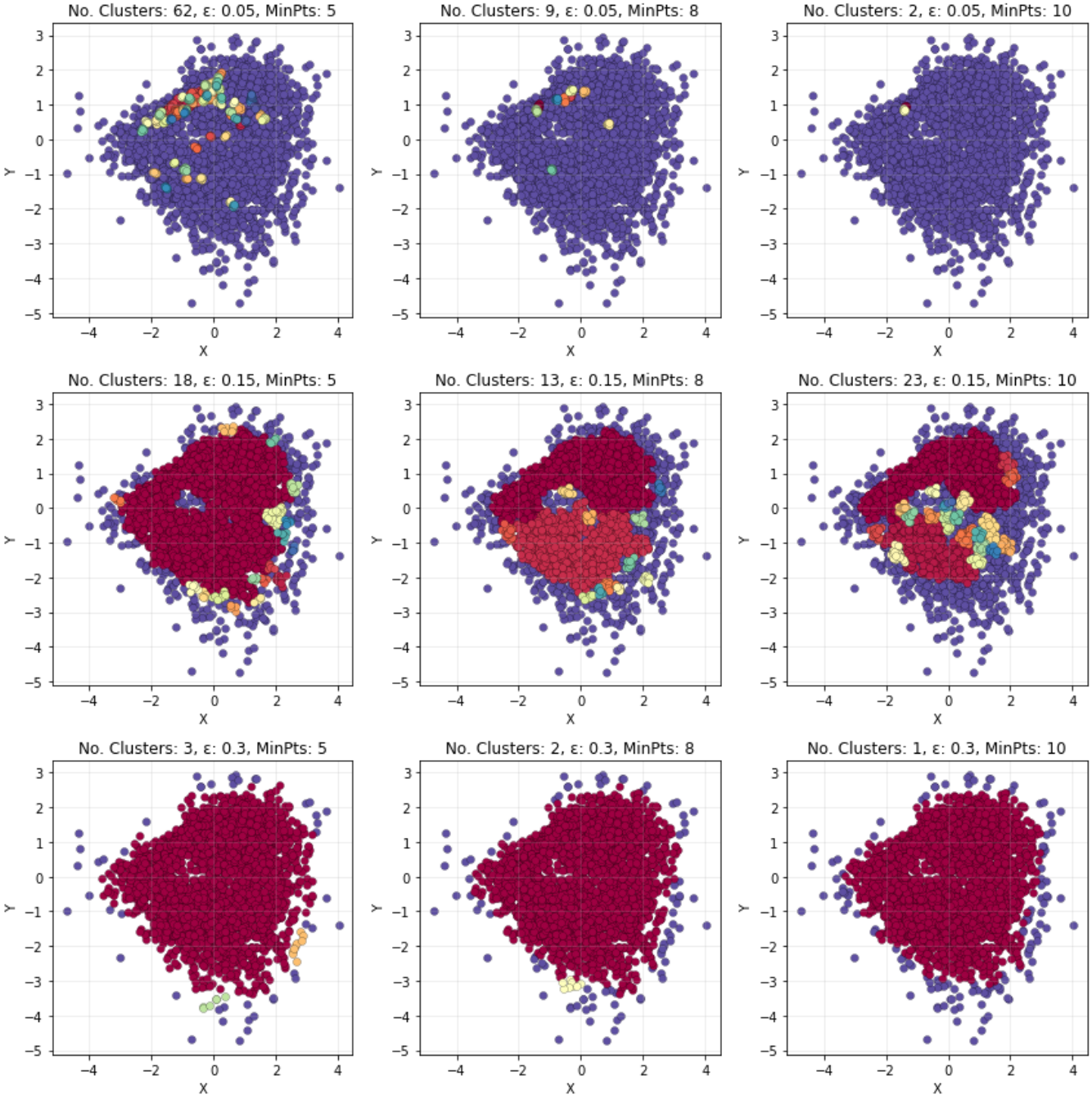
Analysis of k-means++

First, thanks to my use of 10 iterations and k-means++, I feel that these are good enough approximations of the best clusters within the data for each value of k. Regardless, one should be careful about local optima. Anyways. I feel that this algorithm is partitioning the data points in an expected way. Since all of them are cluttered around the center and have no scale nor magnitudes, they are getting sliced like one would cut a cake or a pizza. Thus, I fear that these clusters might not be that meaningful.

In any case, I feel that k-means does not provide enough information to identify the clusters within the data due to the distributions and values of the data points. The points look to be grouped in a single mass, and that might make it difficult for k-means++ to find unique clusters within that ball of points. So, if I were to choose a k value for this data, I think I'll run with either 3 or 4. Based on the elbow plot, those 2 seem to have good improvements for minimizing inertia. In addition, considering that I created this data with 2 clusters per each of 2 classes, I feel that 3 and 4 are closer to the truth. Regarding the other k values, 2 seems to be drastically outperformed by 3 and 4 in terms of inertia, while 5 is starting to produce too many clusters.

DBSCAN

Graph 19 – Clusters from DBSCAN with multiple values of MinPts and Epsilon



Analysis of DBSCAN

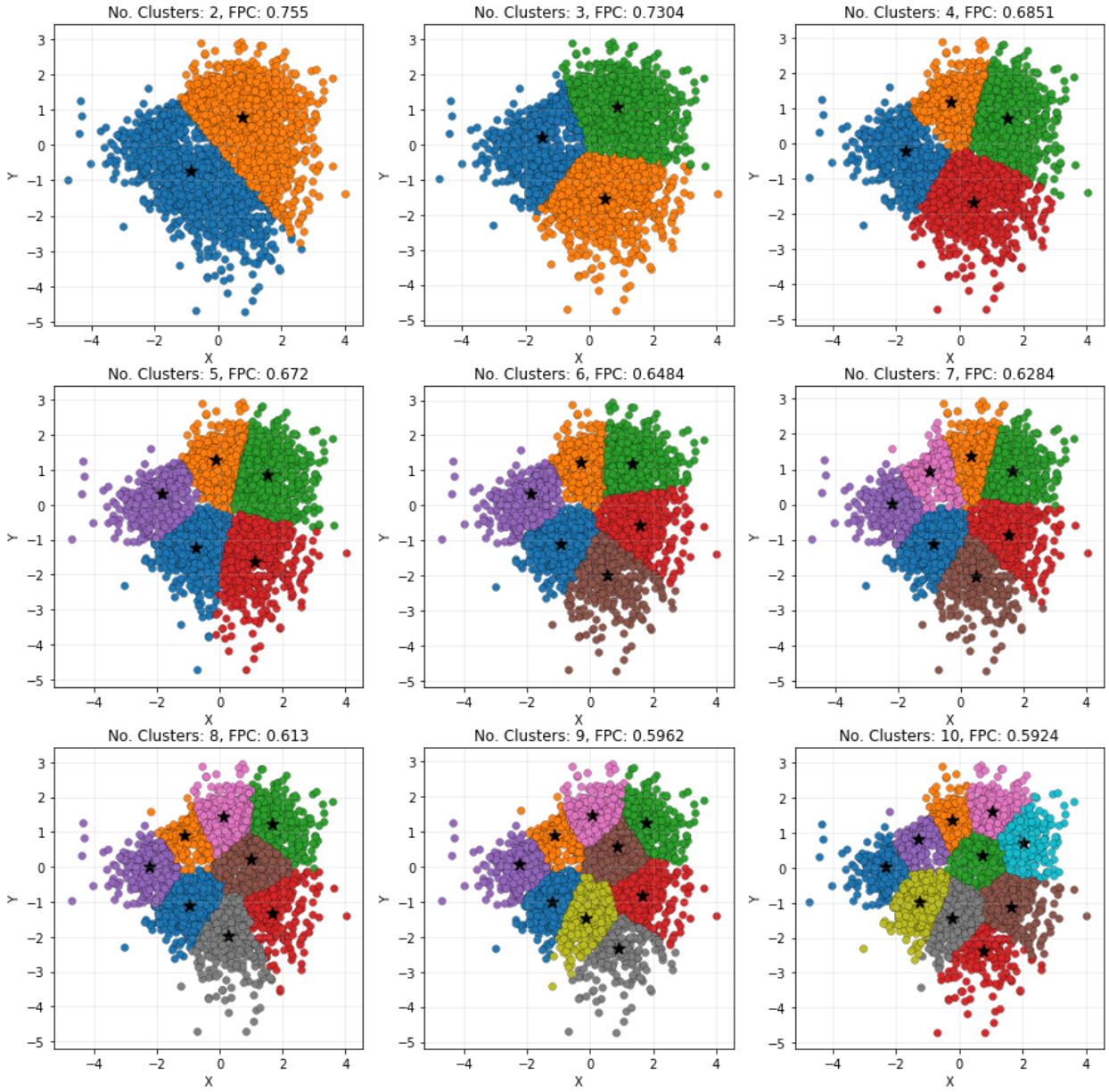
For DBSCAN, I first recognized that its most important parameters were MinPts and Epsilon. As explained in the scikit documentation (n.d.A), while MinPts is important for the algorithm for handling noise, choosing the right value of epsilon is vital for a good run of DBSCAN. A small epsilon won't generate clusters, while a large one would produce a big one (sci-kit, n.d.B); a pattern that I found out as I played with the model.

Based on my testing, I decided to display DBSCAN in a 3x3 matrix with the combinations of 3 values of Epsilon (0.05, 0.15, and 0.3) and 3 values of MinPts (5, 8, 10). I did this because I would be able to perceive how the different combinations of those values would affect my results. It would also give me the chance to compare them easily and identify the better ones. Finally, I also added a code to paint the points in a color spectrum, which would help me to prevent some clusters for having the same color.

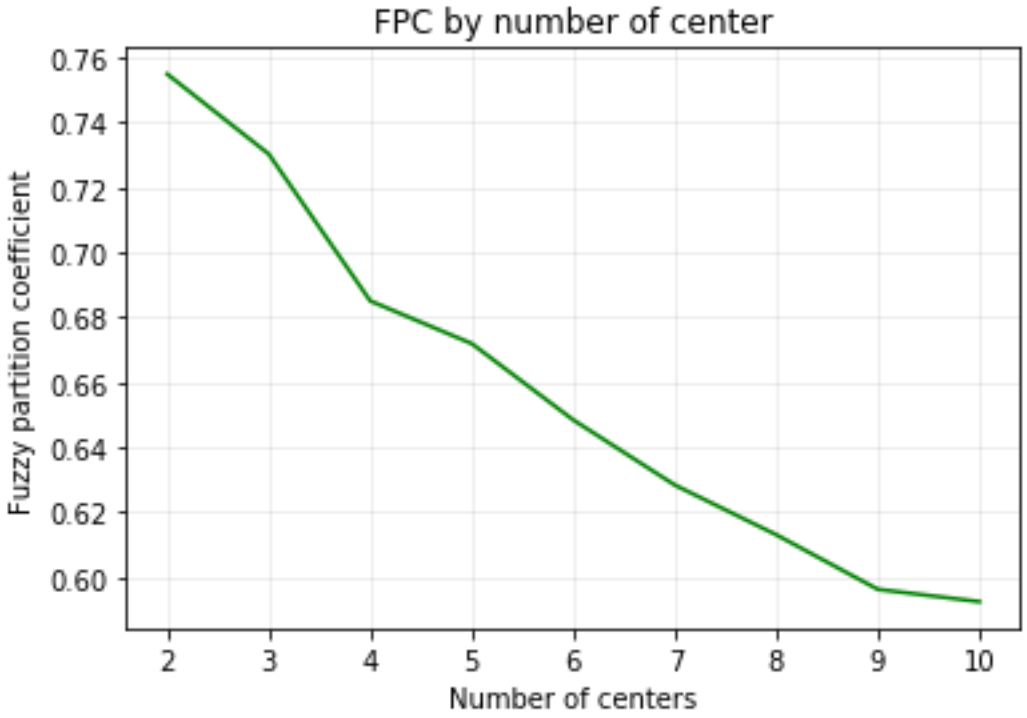
Moving on to the clusters, which are found above, I consider that the best runs of DBSCAN were with an Epsilon of 0.15 and MinPts of either 8 or 10. I conclude this because the other runs do not provide enough information. Runs with Epsilon of 0.05 are not forming clusters, and if they do those clusters are too small. On the other hand, an Epsilon of 0.3 formed a big cluster that provides no information, same as Epsilon 0.15 with MinPts of 5. In contrast, the two runs with Epsilon of 0.15 and MinPts 8 and 10 offer quite a lot of insights. These two runs are indicating that there are two big clusters in the data, with some smaller ones around them. Furthermore, DBSCAN is telling me that the clusters have strange shapes: one as an oval and the other like an axe. This highlights one of the features of DBSCAN: its strength to recognize strange shapes (Xu and Tian, 2015). Based on this, I'll decide to use those two combinations of DBSCAN as the best performing ones.

Fuzzy c-means

Graph 20. Clusters with Fuzzy c-means with value of k and FPC level



Graph 21. Fuzzy Partition Coefficient



Description of Fuzzy c-means

For my third algorithm, I decided to go with Fuzzy c-means. This algorithm is based on fuzzy sets theory: instead of element belonging or not belonging to a group (1 or 0), elements can have a degree of membership to each group (a likelihood from 0 to 1). So, elements can be part of other fuzzy sets (Ross, 2010). In clustering, Fuzzy c-means is a soft clustering algorithm that checks for each point how likely is it for them to be part of each cluster. In contrast, in hard clustering points can only completely belong to one cluster (Kumar, 2021). Some of the benefits of using this method is that you get probabilities rather than exact values, offering a better perspective on the data and that it has a high accuracy; on the flipside, it doesn't work well with higher dimensions, it can fall in local optima, and it is also sensible to the parameters used (Xu and Tian, 2015).

As for the algorithm, Fuzzy c-mean tries to minimize a fuzzy partition matrix, with a parameter m' indicating the degree of fuzziness in the data (Ross, 2010). It does this with heuristics by iteratively optimizing the values of this matrix until it reaches a threshold value or until it reaches a maximum number of iterations (Ross, 2010). To compare different runs of Fuzzy c-mean one can use Dunn's partition coefficient, an indicator that goes from 0 to 1, with higher values often desired since it means that it is closer to a hard solution (NCSS, n.d.)

Analysis of Fuzzy c-means

For its implementation, I based my code on the one found on the scikit-fuzzy python module documentation (n.d.). I decided to run it using an error threshold of 0.005, 1000 maximum iterations, m'= 1.75, seed of 905, and several values of k between 2 and 10. I decided to use a value of m' of 1,75 since it is recommended to be between 1.25 and 2 (Kumar, 2021), and multiple values of k for comparison. Finally, each point was assigned based on the cluster to which they had the largest likelihood of belonging.

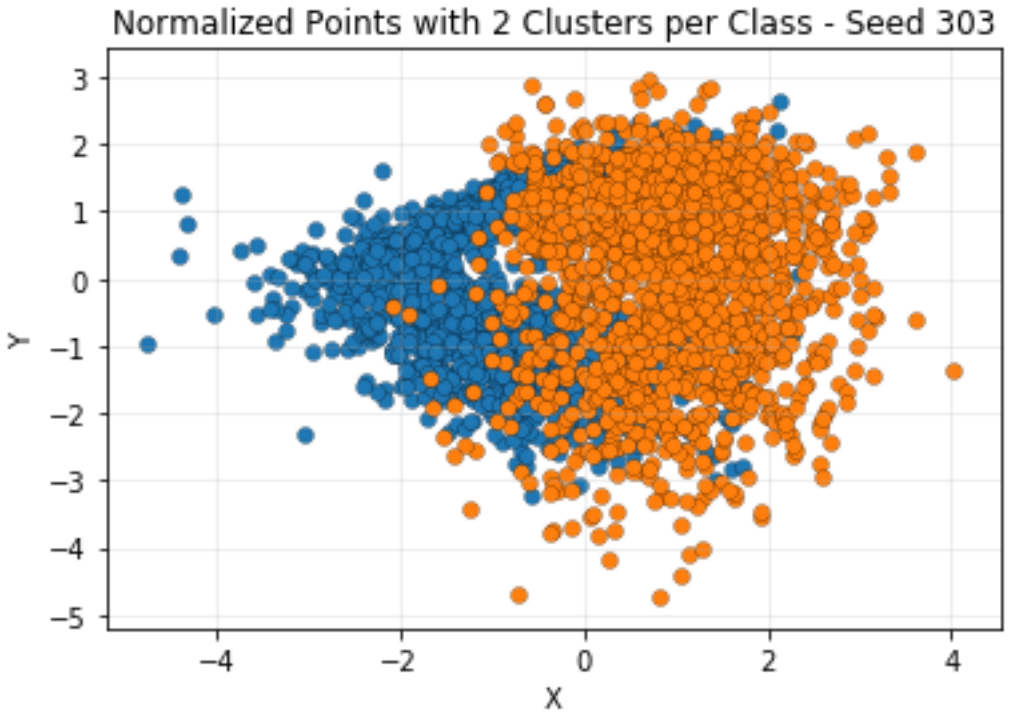
After obtaining my results, I consider that they are quite similar to those obtained using k-means++. This makes sense since the two algorithms are kind of related. In any case, while similar, some of the differences between them lie in the border points, which makes sense since Fuzzy gives you a better perspective on the border points with probabilities (Ross, 2010). Regardless, the results are quite the same, thus they run the risk of being as meaningful as those in k-means, which weren't much.

As for the Fuzzy Partition Coefficient FPC, we see that as k increases the FPC decreases as well, with some changes in the slope here and then. Based on the FPC chart, the clusters I'm paying the more attention are those with k-values of 3 and 4. Based on what I know from the original data, 3 and 4 make sense since there are 4 clusters in there divided in two classes. Plus, they have relatively high FPC values, and the slope changes noticeably after 4. Thus, based on my Fuzzy results, I'd use clusters 3 and 4.

2.4 Final Comparison of the three algorithms

To begin this section, I'd like to show the actual cluster patter in the 3000-points dataset, which is in the graph below. As a reminder, there are supposed to be 2 clusters in each of the classes, denoted by colors. As one can see, there are some interesting things going on in the dataset. First, we have the shape of the clusters. While the blue points form some noticeable clusters in the shape of ovals, the orange ones don't. This meant that the algorithms had a hard time detecting the clusters due to the shape. Furthermore, the blue and orange points overlap quite a lot. This is one of the reasons why I chose this dataset: I wanted to see how the algorithms handle this overlapping clusters.

Graph 22. Classes in the 3000-points dataset



Comparing the algorithms, for this data, I'd rank them in the following order from best to worst: DBSCAN, Fuzzy c-means, and k-means++. Overall, I find that the closer approximation came from DBSCAN with Epsilon = 0.15, and MinPts of 10, resulting in 23 clusters. I feel that is not even close because DBSCAN was the only algorithm that captured the shape of the clusters from the blue points within the data. I feel that this happened because this algorithm can handle strange cluster shapes unlike the others, and the make_classification function tends to produce clusters in those forms. Still, DBSCAN was limited when it came to recognizing the clusters from the orange points, but all of them were lacking in this aspect.

Talking about the orange points, I think that the points that overlap from the blue and orange classes should start being labeled as their own cluster. Since all the data is being analyzed at the same time, it's not reasonable to separate those points even if they don't come from the same

classes. Thus, we can think of that overlap as a new class or cluster of its own: the overlap between orange and blue. In that sense even if make_classification was supposed to produce 4 clusters, the resulting configuration ended up with a different number of clusters. Regardless, DBSCAN was still the sole algorithm that I used that detected the shapes of the blue points, still leaving it on top.

Thought, considering everything, there's another cluster result that caught my attention. In the runs of Fuzzy c-means with high values of k (9 and 10), the shape of the blue points starts to appear. Referring to k = 10, the blue, purple, orange, and pink sections form one oval, while the blue, yellow, gray, and red clusters form another of the blue ovals. Then, some of the clusters depict overlapping parts: the orange, pink, gray and red clusters mark zones where blue and orange points overlap. Furthermore, the green, cyan, and brown sections are zones with mostly orange points. I had already read from the literature that supposedly Fuzzy c-means can handle overlap adequately (Kumar, 2021), but after thinking more about my results with k = 10, I can start to see how. True, it might have not detected them on lower values. However, with the degree of overlap between the two classes, it's surprising that it caught wind of that overlap even if it was at higher values of k.

For the remaining algorithm, that is k-means++, I think that in this scenario is not the best performing for many reasons. First, it naturally gets outclassed by Fuzzy c-means since they are very similar, but Fuzzy excels by offering probabilities. Second, the way the data is distributed is detrimental to the algorithm since k-means seems to favor data that has a much clearer visual pattern. Still, it is worth mentioning that maybe with higher values of k this algorithm could produce a more interesting result like Fuzzy did. Also, one shouldn't forget how easy and cheap to implement k-means++ is.

To close this comparison, I feel that the main lesson I can take is that clustering is not something that you can produce with one run, and maybe even 10 or 100, and with just one algorithm. In this last exercise, I was having some troubles finding the best results of a dataset that I already knew the shape of. Imagine for a data that you don't know anything about or that has higher dimensions. That's why I am starting to think of clustering as something that not only needs science but also a lot of common sense. Data scientists don't want to end up with just one gigantic cluster or with hundreds of little ones. They want clusters that are meaningful more than anything, regardless of how low or high the values of the parameters can be. Only with careful knowledge about the data and with the proper visualizations we can use our common sense along with any advanced algorithm to discover the clusters hidden within the data.

References

Kumar, S. (2021) *Fuzzy C-Means Clustering —Is it Better than K-Means Clustering?* Towards Data Science. Available at: https://towardsdatascience.com/fuzzy-c-means-clustering-is-it-better-than-k-means-clustering-448a0aba1ee7 Accessed (18 March 2022)

NCSS (n.d.) *Fuzzy Clustering.* NCSS. Available at: https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Fuzzy_Clustering.pdf Accessed (18 March 2022)

Raune, E. (2022) Example Jupyter Notebook for Assignment 2

Random.org (n.d.) *Random.org*. Available at: https://www.random.org/ Accessed (15 March 2022)

Ross, T. (2010) *Fuzzy Logic with Engineering Applications.* John Wiley & Sons. Available at: http://iauctb.ac.ir/Files/%D9%88%D8%A8%20%D8%B3%D8%A7%DB%8C%D8%AA%20%D8%A7%D8%B3%D8%A7%D8%AA%DB%8C%D8%AF/fuzzy%20logic%20with%20engineering%20application-3rdEdition.pdf Accessed (18 March 2022)

scikit-fuzzy (n.d.) *Fuzzy c-means clustering*. Available at: https://pythonhosted.org/scikit-fuzzy/auto_examples/plot_cmeans.html Accessed (18 March 2022)

scikit -learn (n.d.A) *Demo of DBSCAN clustering algorithm.* Available at: https://scikit-learn.org/stable/auto_examples/cluster/plot_dbscan.html#sphx-glr-auto-examples-cluster-plot-dbscan-py Accessed (19 March 2022)

scikit - learn (n.d.B) *Clustering*. Available at: https://scikit-learn.org/stable/modules/clustering.html#k-means Accessed (15 March 2022)

scikit - learn (n.d.C) *sklearn.cluster.KMeans*. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans Accessed (15 March 2022)

scikit - learn (n.d.D) *sklearn.datasets.make_classification*. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans Accessed (15 March 2022)

Woodcock, H. (2021) *Stop using numpy.random.seed().* Towards Data Science. Available at: https://towardsdatascience.com/stop-using-numpy-random-seed-581a9972805f Accessed (15 March 2022)

Xu, D. & Tian, Y. (2015) "A Comprehensive Survey of Clustering Algorithms". *Ann. Data. Sci*., 2, pp. 165-193. Available at: https://link.springer.com/content/pdf/10.1007/s40745-015-0040-1.pdf Accessed (15 March 2022)