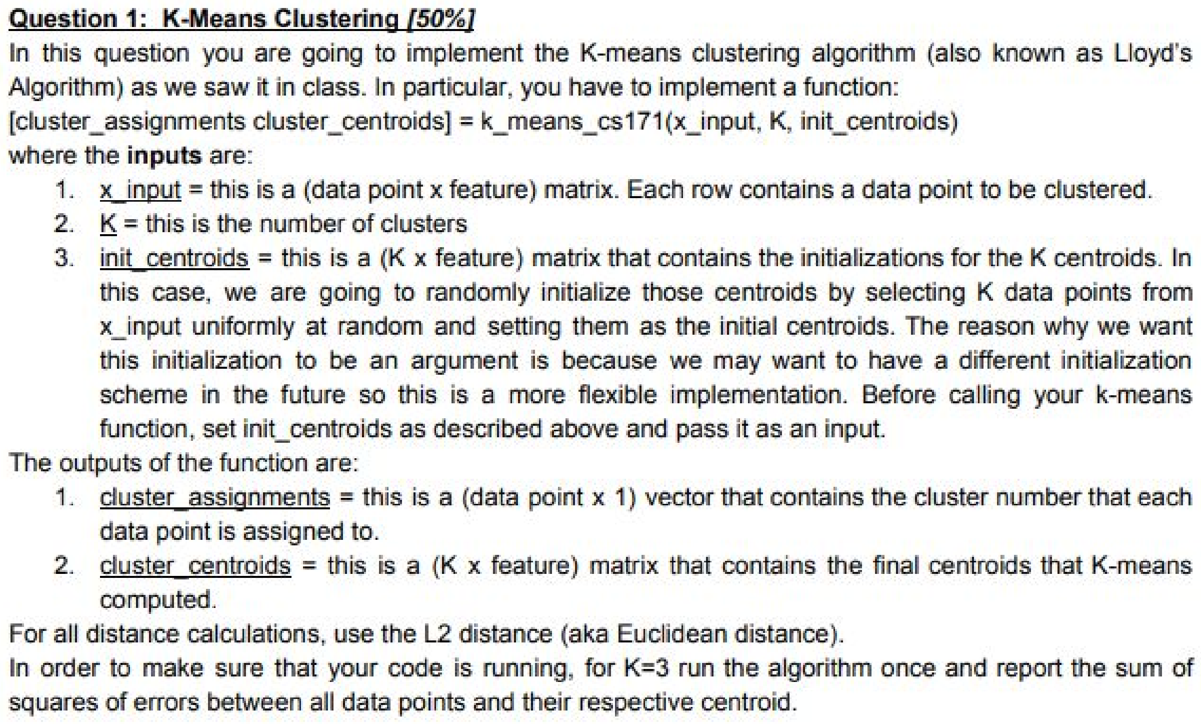
Brennan Hall
861198641
CS171 - 18S
Late Days Used: 0
Total Late Days: 2

Assignment 3



Question 1: K-Means Clustering *[50%]*

In this question you are going to implement the K-means clustering algorithm (also known as Lloyd's Algorithm) as we saw it in class. In particular, you have to implement a function:
[cluster_assignments cluster_centroids] = k_means_cs171(x_input, K, init_centroids)
where the **inputs** are:

1. x_input = this is a (data point x feature) matrix. Each row contains a data point to be clustered.
2. K = this is the number of clusters
3. init_centroids = this is a (K x feature) matrix that contains the initializations for the K centroids. In this case, we are going to randomly initialize those centroids by selecting K data points from x_input uniformly at random and setting them as the initial centroids. The reason why we want this initialization to be an argument is because we may want to have a different initialization scheme in the future so this is a more flexible implementation. Before calling your k-means function, set init_centroids as described above and pass it as an input.

The outputs of the function are:

1. cluster_assignments = this is a (data point x 1) vector that contains the cluster number that each data point is assigned to.
2. cluster_centroids = this is a (K x feature) matrix that contains the final centroids that K-means computed.

For all distance calculations, use the L2 distance (aka Euclidean distance).
In order to make sure that your code is running, for K=3 run the algorithm once and report the sum of squares of errors between all data points and their respective centroid.

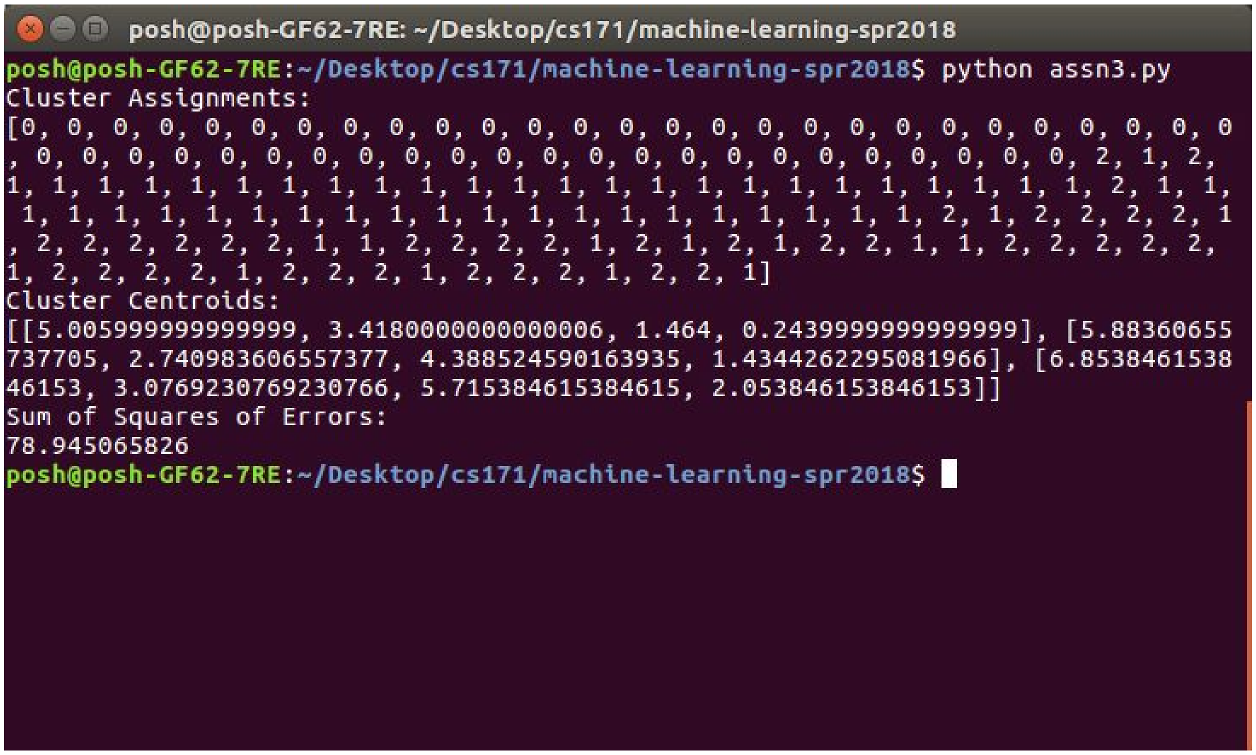
When approaching K-Means Clustering, I sought to understand each part of the algorithm in isolation. The process itself is relatively simple but its implementation can prove confusing if went about in the wrong fashion. In this sense, I unit tested each component of the K-Means algorithm on its own, making sure that the formation and manipulation of clusters and centroids was happening precisely as intended.

My k-means function starts by being passed in the iris data points, a value for K, and a sample of data points as the initial centroids. The main loop starts by finding the nearest centroid and assigning it to that centroid's cluster. I decided to implement a closest_centroid function that determines this. This calculation uses Euclidean distance.

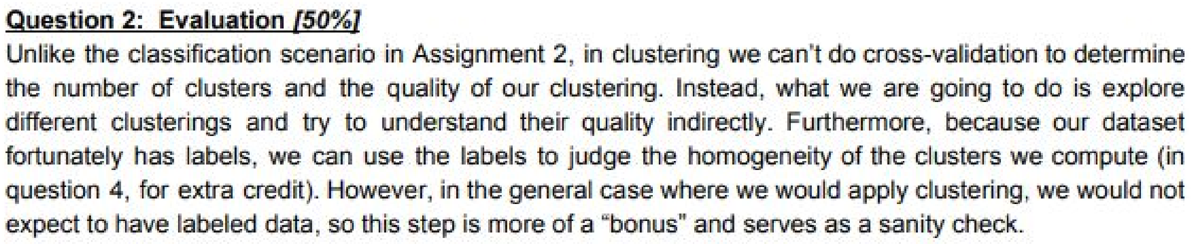
Then for each cluster, a new centroid is calculated as the mean of all points within the cluster. This was perhaps the most complicated nested loop for me to implement, but naming conventions made it easy to work through the complexity. At the end of the loop, there is a check to see if the centroids have shifted. If they have, then the main loop continues. Otherwise, the clusters and point assignments are returned.

With the random nature of k-means, there is occasionally an edge case in which a centroid has no points associated with it. In this instance, taking the mean creates a divide-by-zero error. Nevertheless, I handle this case by setting the mean to 0 without any illegal calculation.

Below is a completed run-through of my k-means algorithm, including the sum of squared errors. Most always the SSE is similar to the value shown below, but occasionally the function gets “unlucky” and the clusters get grouped only to 2 centroids, which causes a much higher SSE.



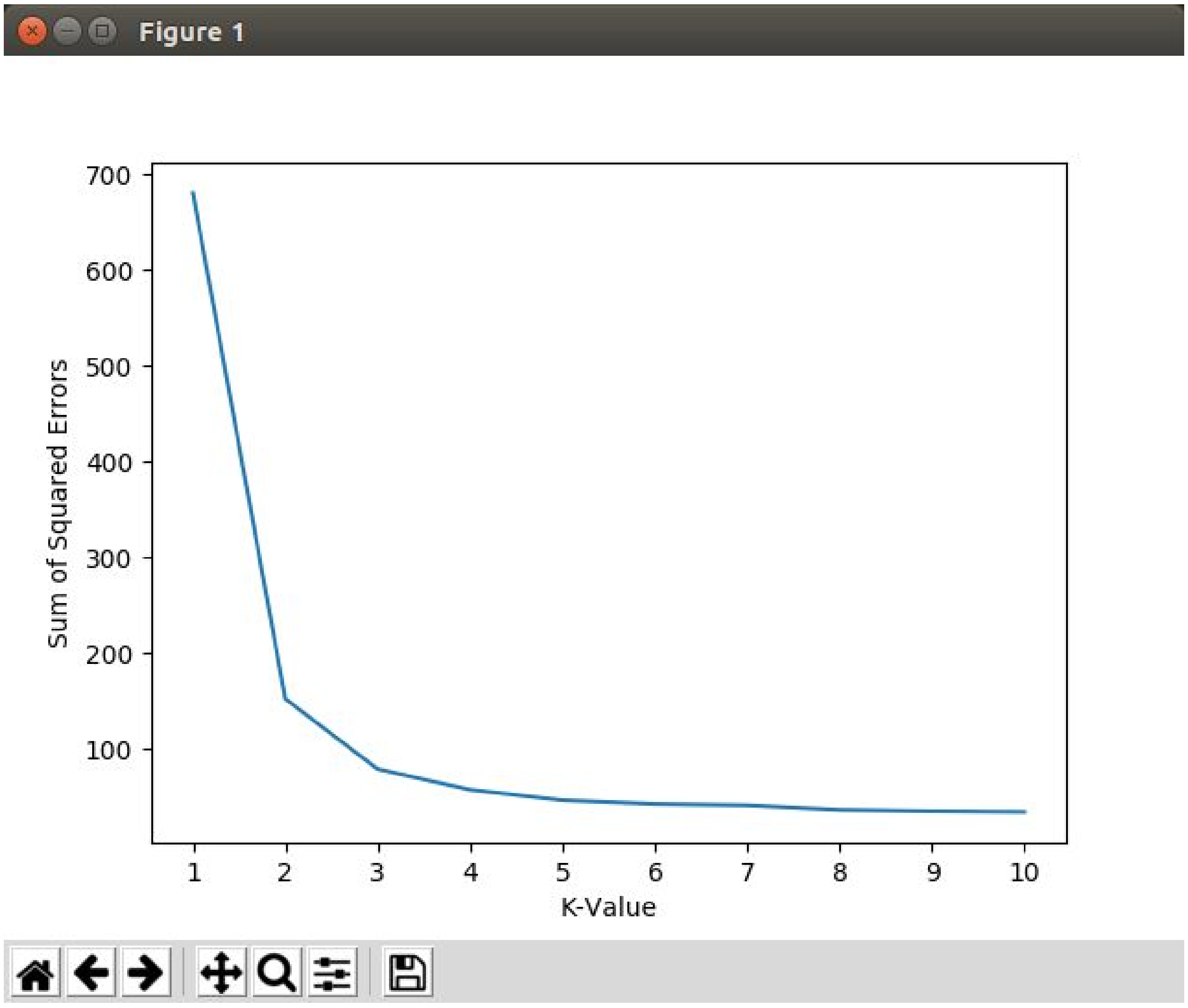
```
posh@posh-GF62-7RE: ~/Desktop/cs171/machine-learning-spr2018
posh@posh-GF62-7RE:~/Desktop/cs171/machine-learning-spr2018$ python assn3.py
Cluster Assignments:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 2,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 2, 2, 1
, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2, 2, 2, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 2, 2, 2, 2,
1, 2, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 1]
Cluster Centroids:
[[5.005999999999999, 3.4180000000000006, 1.464, 0.2439999999999999], [5.88360655
737705, 2.740983606557377, 4.388524590163935, 1.4344262295081966], [6.8538461538
46153, 3.0769230769230766, 5.715384615384615, 2.053846153846153]]
Sum of Squares of Errors:
78.945065826
posh@posh-GF62-7RE:~/Desktop/cs171/machine-learning-spr2018$ 
```



Question 2: Evaluation *[50%]*

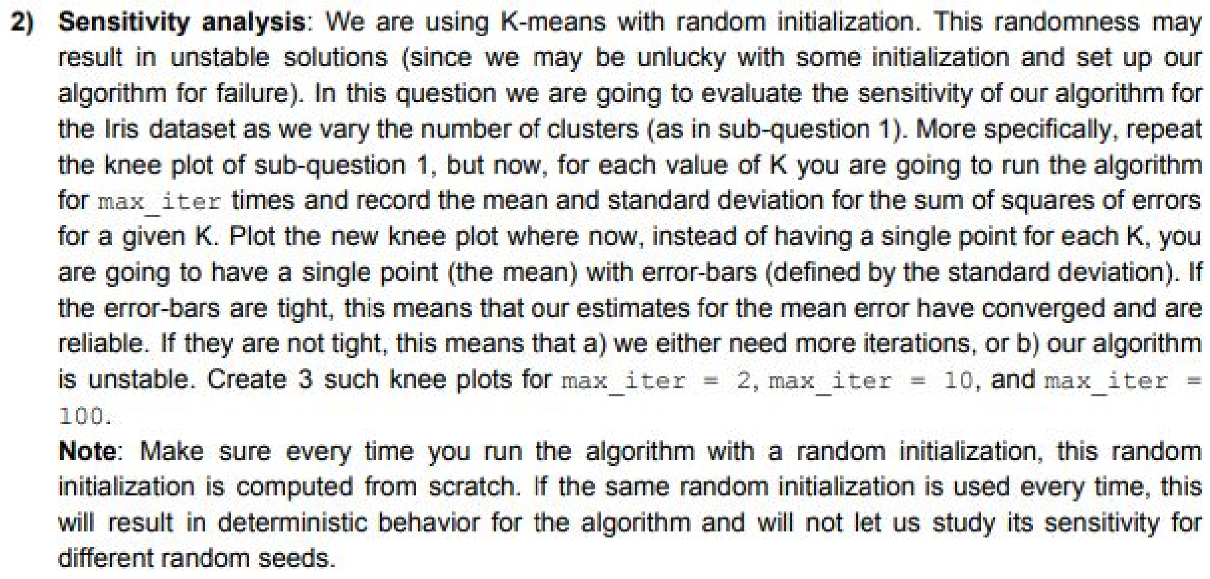
Unlike the classification scenario in Assignment 2, in clustering we can't do cross-validation to determine the number of clusters and the quality of our clustering. Instead, what we are going to do is explore different clusterings and try to understand their quality indirectly. Furthermore, because our dataset fortunately has labels, we can use the labels to judge the homogeneity of the clusters we compute (in question 4, for extra credit). However, in the general case where we would apply clustering, we would not expect to have labeled data, so this step is more of a “bonus” and serves as a sanity check.

1) **Knee Plot**: As we saw in class, a good heuristic for judging the quality of K-means clustering and finding the number of clusters is the "knee plot". The name of the plot comes from the "knee" in the line that we draw, indicating the point where the error stops improving dramatically, which, in turn, may indicate a good number of clusters. In this question you should generate the knee plot, i.e., the sum of squares of errors of all data points from their assigned cluster on the y-axis (i.e., the function that K-means is minimizing) as a function of the number of clusters K, for K taking values in `[1 2 3 4 5 6 7 8 9 10]` on the x-axis. Make the plot after running the algorithm once. At which K does the "knee" appear? Does it agree with the number of classes in the Iris dataset?



After running the graph many times, the graph ends up at this shape or something similar the vast majority of the time. It is clear to see that the "knee" of the graph is at K=3. This means that improvement is very minimal for larger values of K, which denotes that it does in fact agree with the true number of Iris classes.

This makes sense considering that there are 3 pre-determined classes in the iris dataset, and the mean values for each of these classes are significantly distinct as we saw during Assignment 1. In this sense, the natural inclination for a functioning k-means algorithm would be to recreate this distinction for the given number of classes and not exceed it.



2) **Sensitivity analysis**: We are using K-means with random initialization. This randomness may result in unstable solutions (since we may be unlucky with some initialization and set up our algorithm for failure). In this question we are going to evaluate the sensitivity of our algorithm for the Iris dataset as we vary the number of clusters (as in sub-question 1). More specifically, repeat the knee plot of sub-question 1, but now, for each value of K you are going to run the algorithm for `max_iter` times and record the mean and standard deviation for the sum of squares of errors for a given K. Plot the new knee plot where now, instead of having a single point for each K, you are going to have a single point (the mean) with error-bars (defined by the standard deviation). If the error-bars are tight, this means that our estimates for the mean error have converged and are reliable. If they are not tight, this means that a) we either need more iterations, or b) our algorithm is unstable. Create 3 such knee plots for `max_iter = 2`, `max_iter = 10`, and `max_iter = 100`.
Note: Make sure every time you run the algorithm with a random initialization, this random initialization is computed from scratch. If the same random initialization is used every time, this will result in deterministic behavior for the algorithm and will not let us study its sensitivity for different random seeds.

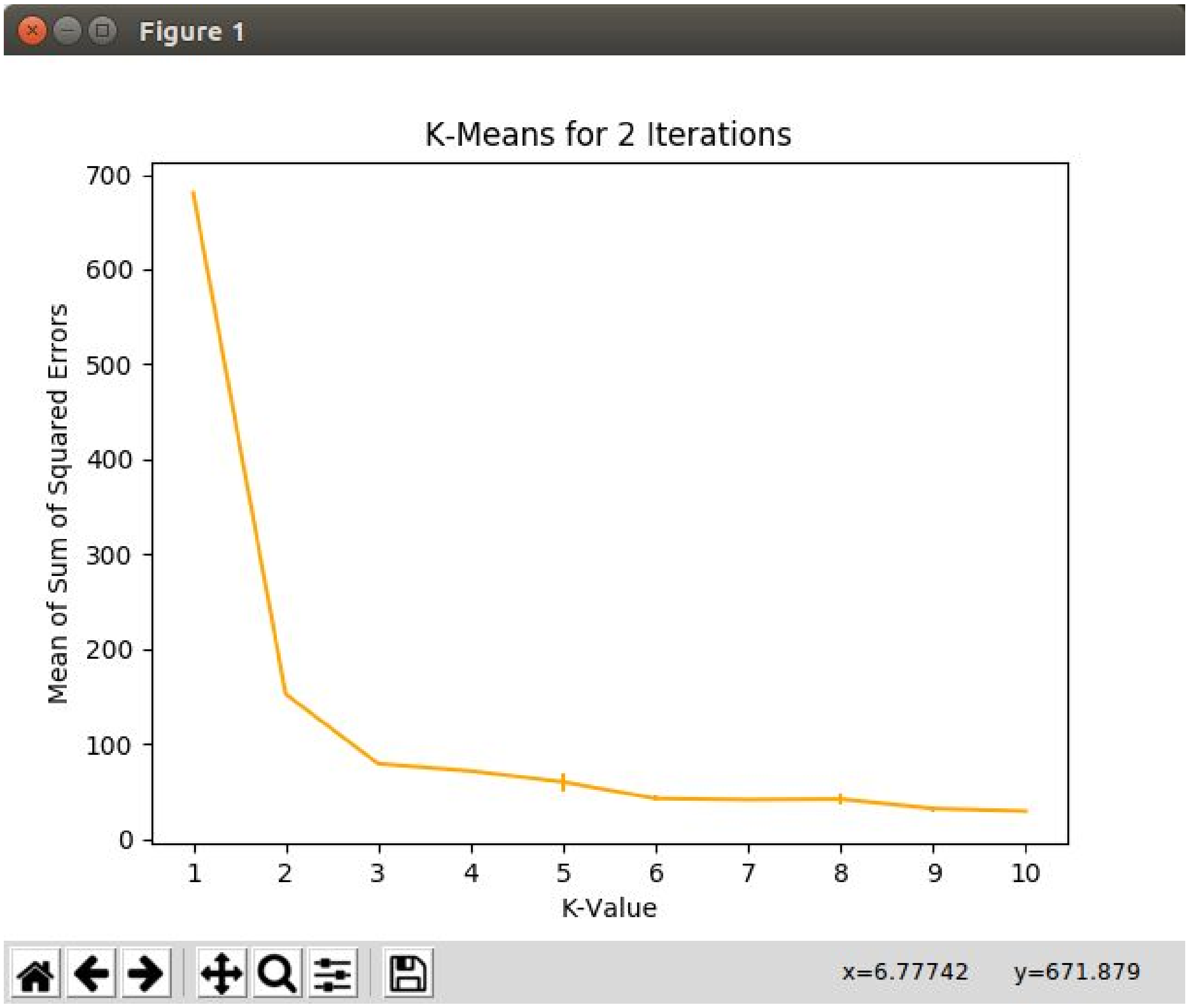


Figure 1
K-Means for 2 Iterations
Mean of Sum of Squared Errors
K-Value
0
100
200
300
400
500
600
700
1
2
3
4
5
6
7
8
9
10
x=6.77742 y=671.879

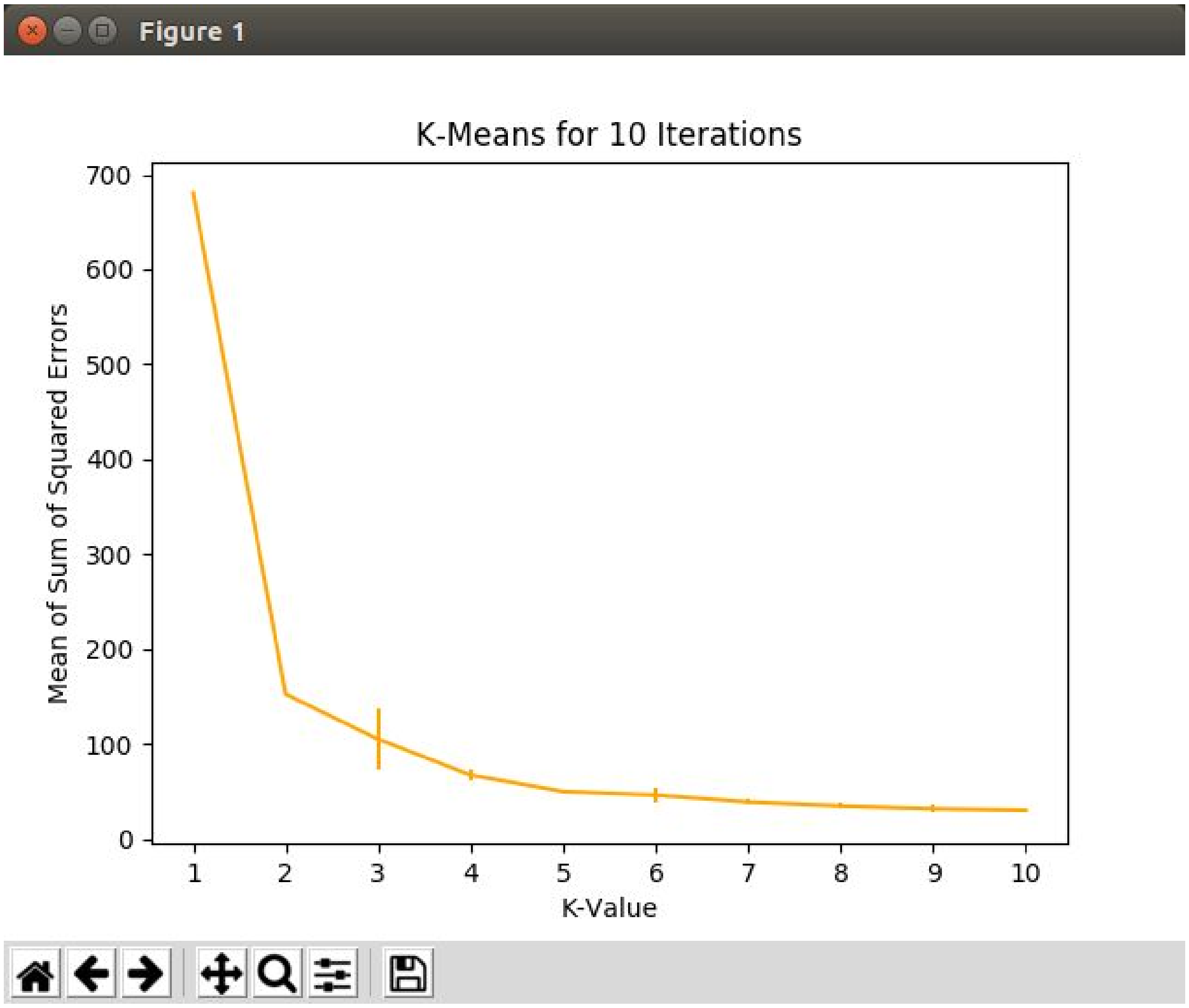
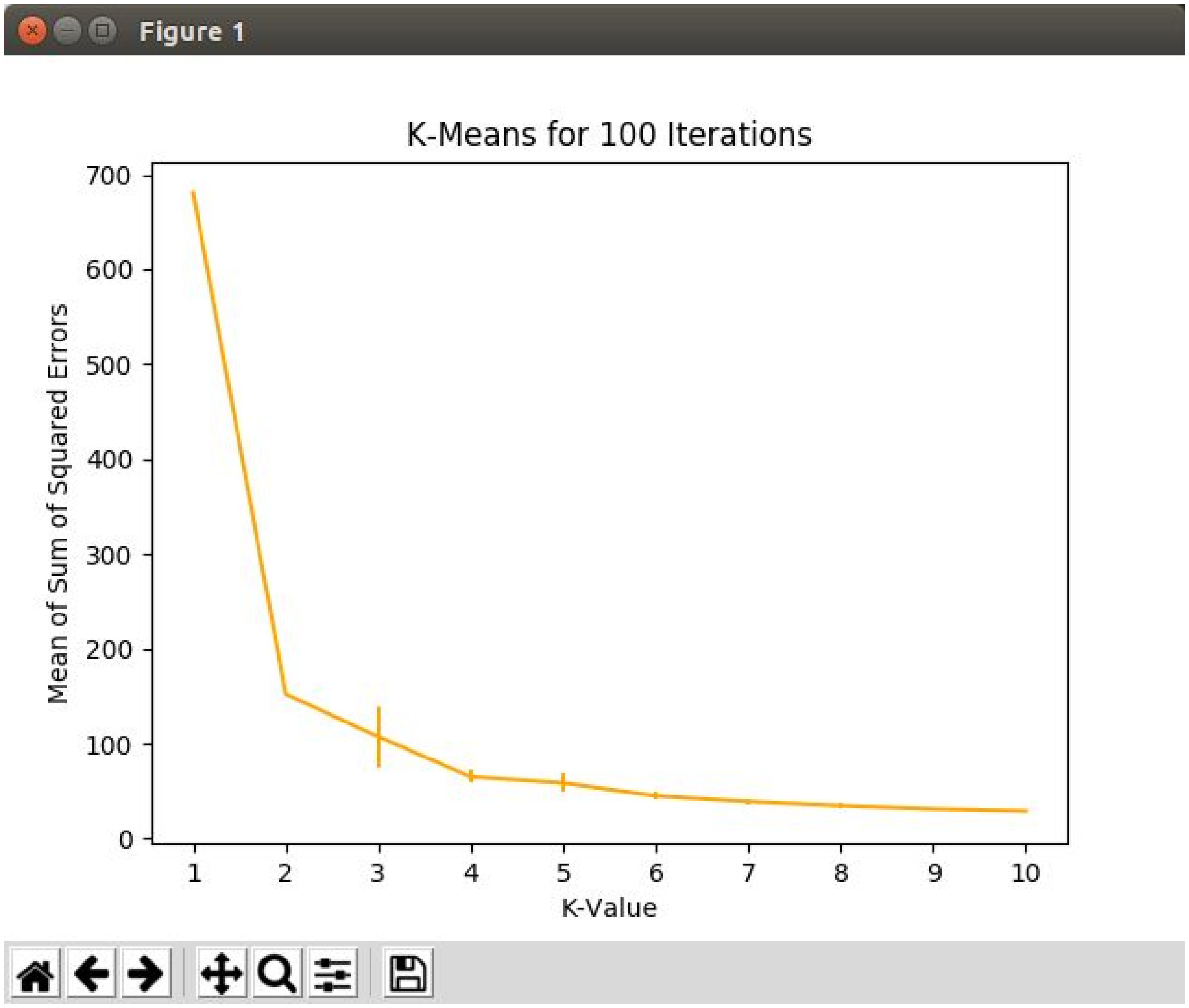
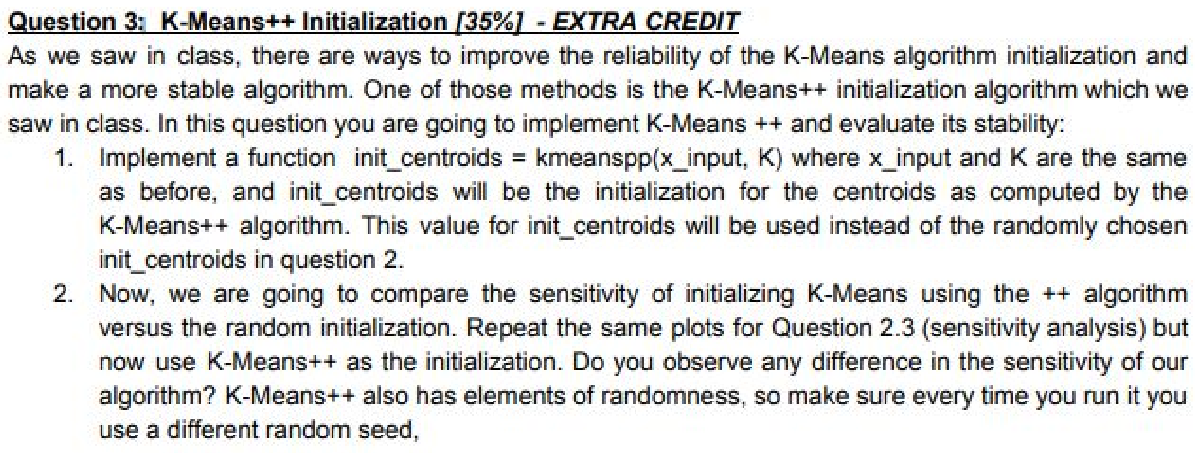


Figure 1
K-Means for 10 Iterations
Mean of Sum of Squared Errors
K-Value
0
100
200
300
400
500
600
700
1
2
3
4
5
6
7
8
9
10



As per the requirements, I made sure that each iteration of the algorithm was randomly initialized by using a completely different random seed each time. Nevertheless, running k-means for more than one iteration gives valuable information about the algorithm. The first graph looks nearly identical to the original 1 iteration of k-means and has very, very small error bars. However, the other graphs have more noticable standard deviations. The highest standard deviation is always when K=3, meaning that the means differ on a large level when run multiple times.

In this sense, K=3 is trouble for the algorithm in regards to certain data points that may be on the fence of two different classes. Certain random initializations get it right, as per the low end of the standard deviation, but others fail, as per the high end. When there more iterations, standard deviations for higher values of K are more noticeable, as is standard of an algorithm being run for so many times.

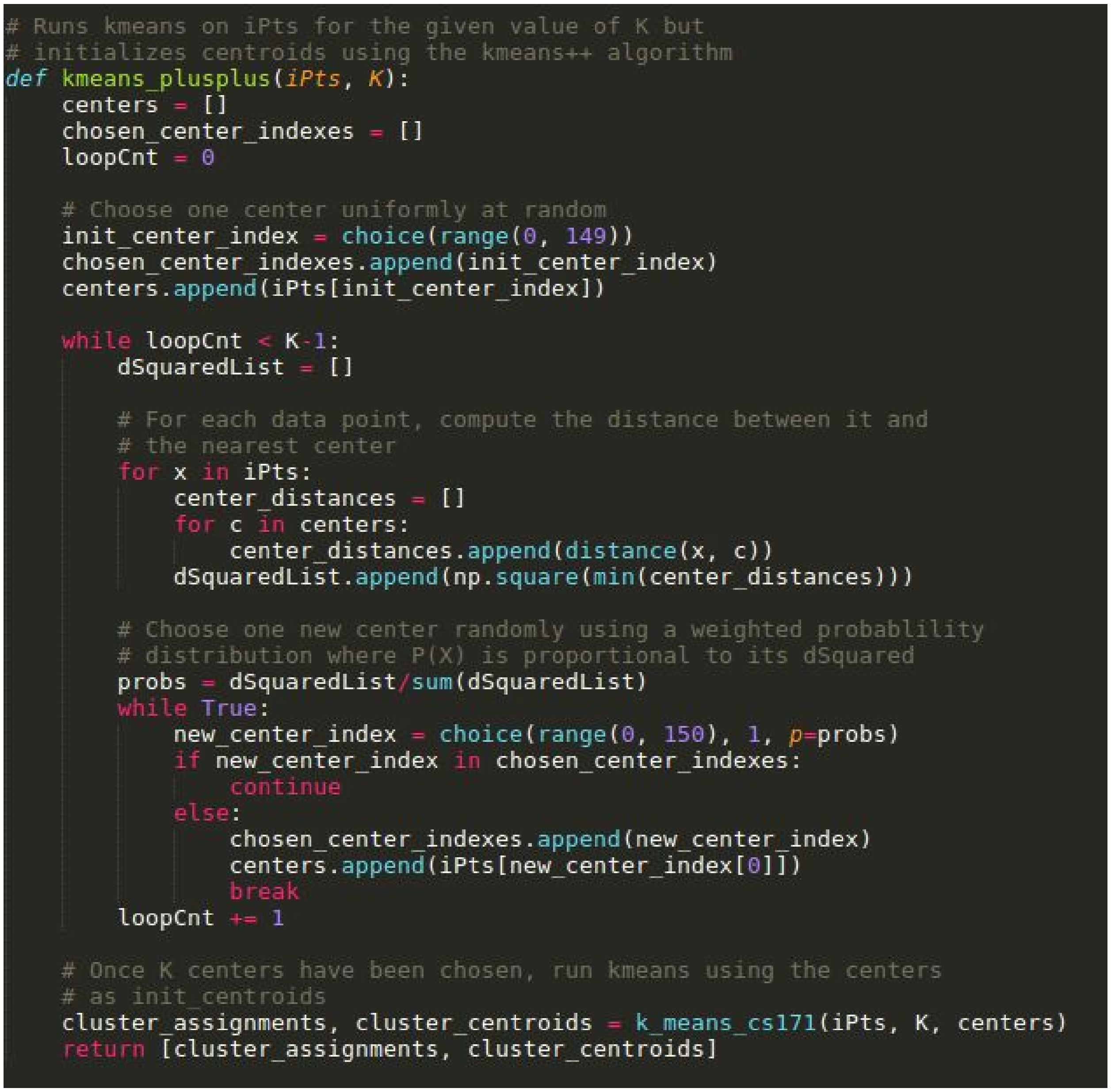


Question 3: K-Means++ Initialization *[35%] - EXTRA CREDIT*

As we saw in class, there are ways to improve the reliability of the K-Means algorithm initialization and make a more stable algorithm. One of those methods is the K-Means++ initialization algorithm which we saw in class. In this question you are going to implement K-Means ++ and evaluate its stability:

1. Implement a function init_centroids = kmeanspp(x_input, K) where x_input and K are the same as before, and init_centroids will be the initialization for the centroids as computed by the K-Means++ algorithm. This value for init_centroids will be used instead of the randomly chosen init_centroids in question 2.
2. Now, we are going to compare the sensitivity of initializing K-Means using the ++ algorithm versus the random initialization. Repeat the same plots for Question 2.3 (sensitivity analysis) but now use K-Means++ as the initialization. Do you observe any difference in the sensitivity of our algorithm? K-Means++ also has elements of randomness, so make sure every time you run it you use a different random seed,

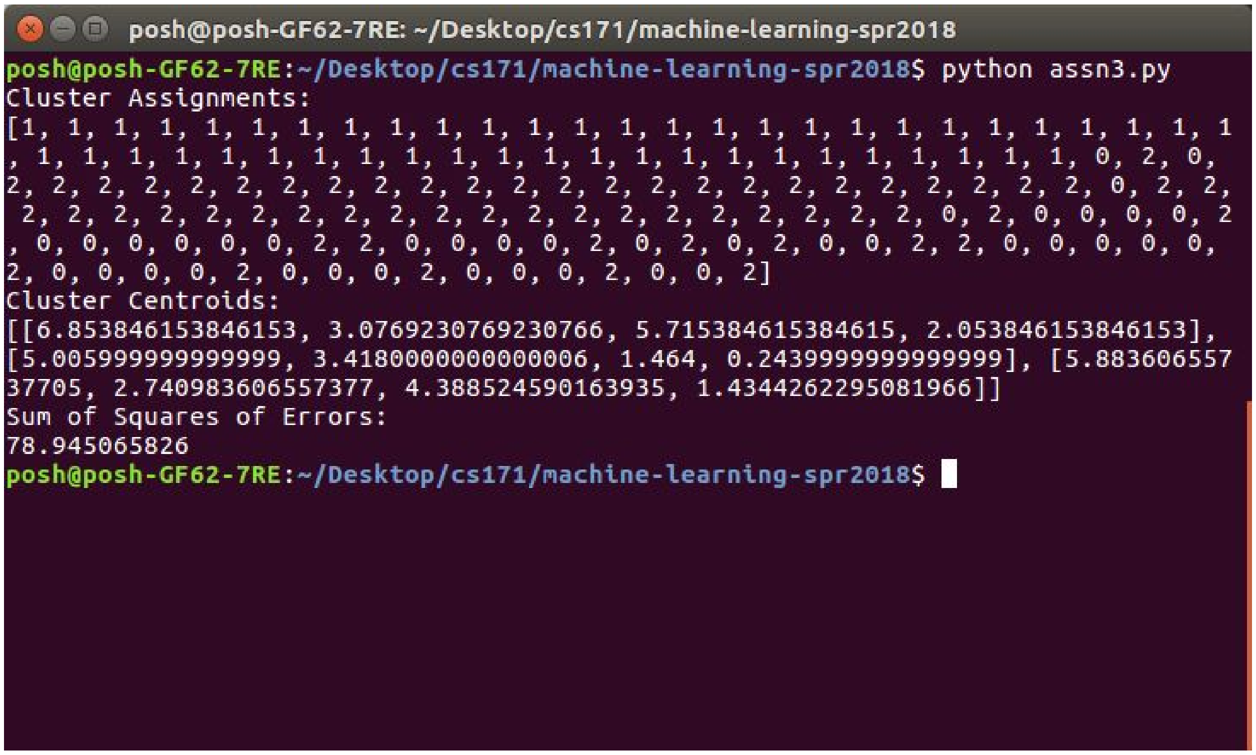
Understanding K-Means++ was very simple, since the hard computation is already done in the standard k-means function. In this sense, K-Means++ is just a smarter way to initialize the centroids for k-means which makes the centroids settle faster and lowers the overall runtime of the main algorithm. It was important for me to realize that K-means++ doesn't necessarily increase the precision of K-Means, it simply makes it run quicker.



```python
# Runs kmeans on iPts for the given value of K but
# initializes centroids using the kmeans++ algorithm
def kmeans_plusplus(iPts, K):
    centers = []
    chosen_center_indexes = []
    loopCnt = 0

    # Choose one center uniformly at random
    init_center_index = choice(range(0, 149))
    chosen_center_indexes.append(init_center_index)
    centers.append(iPts[init_center_index])

    while loopCnt < K-1:
        dSquaredList = []

        # For each data point, compute the distance between it and
        # the nearest center
        for x in iPts:
            center_distances = []
            for c in centers:
                center_distances.append(distance(x, c))
            dSquaredList.append(np.square(min(center_distances)))

        # Choose one new center randomly using a weighted probablility
        # distribution where P(X) is proportional to its dSquared
        probs = dSquaredList/sum(dSquaredList)
        while True:
            new_center_index = choice(range(0, 150), 1, p=probs)
            if new_center_index in chosen_center_indexes:
                continue
            else:
                chosen_center_indexes.append(new_center_index)
                centers.append(iPts[new_center_index[0]])
                break
        loopCnt += 1

    # Once K centers have been chosen, run kmeans using the centers
    # as init_centroids
    cluster_assignments, cluster_centroids = k_means_cs171(iPts, K, centers)
    return [cluster_assignments, cluster_centroids]
```

As shown in my implementation above, the sole improvement that K-Means++ brings to the table is the way in which init_centroids is evaluated. At the end of my implementation, I simply call the standard k-means algorithm with the newly computed centroids from the computation above. The basis of K-Means++ is built upon the notion that centroids should have a

The data below was provided by a run through of K-Means++ with K=3, which is nearly identical to the data provided from my original first run of regular K-Means. This once again reiterates that K-Means++ does not boost accuracy, but merely improves runtime.



```
posh@posh-GF62-7RE: ~/Desktop/cs171/machine-learning-spr2018
posh@posh-GF62-7RE:~/Desktop/cs171/machine-learning-spr2018$ python assn3.py
Cluster Assignments:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 2, 0,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 0, 0, 0, 0, 2
, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 2, 0, 2, 0, 2, 0, 0, 2, 2, 0, 0, 0, 0, 0,
2, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 2]
Cluster Centroids:
[[6.853846153846153, 3.0769230769230766, 5.715384615384615, 2.053846153846153],
[5.005999999999999, 3.4180000000000006, 1.464, 0.2439999999999999], [5.883606557
37705, 2.740983606557377, 4.388524590163935, 1.4344262295081966]]
Sum of Squares of Errors:
78.945065826
posh@posh-GF62-7RE:~/Desktop/cs171/machine-learning-spr2018$
```

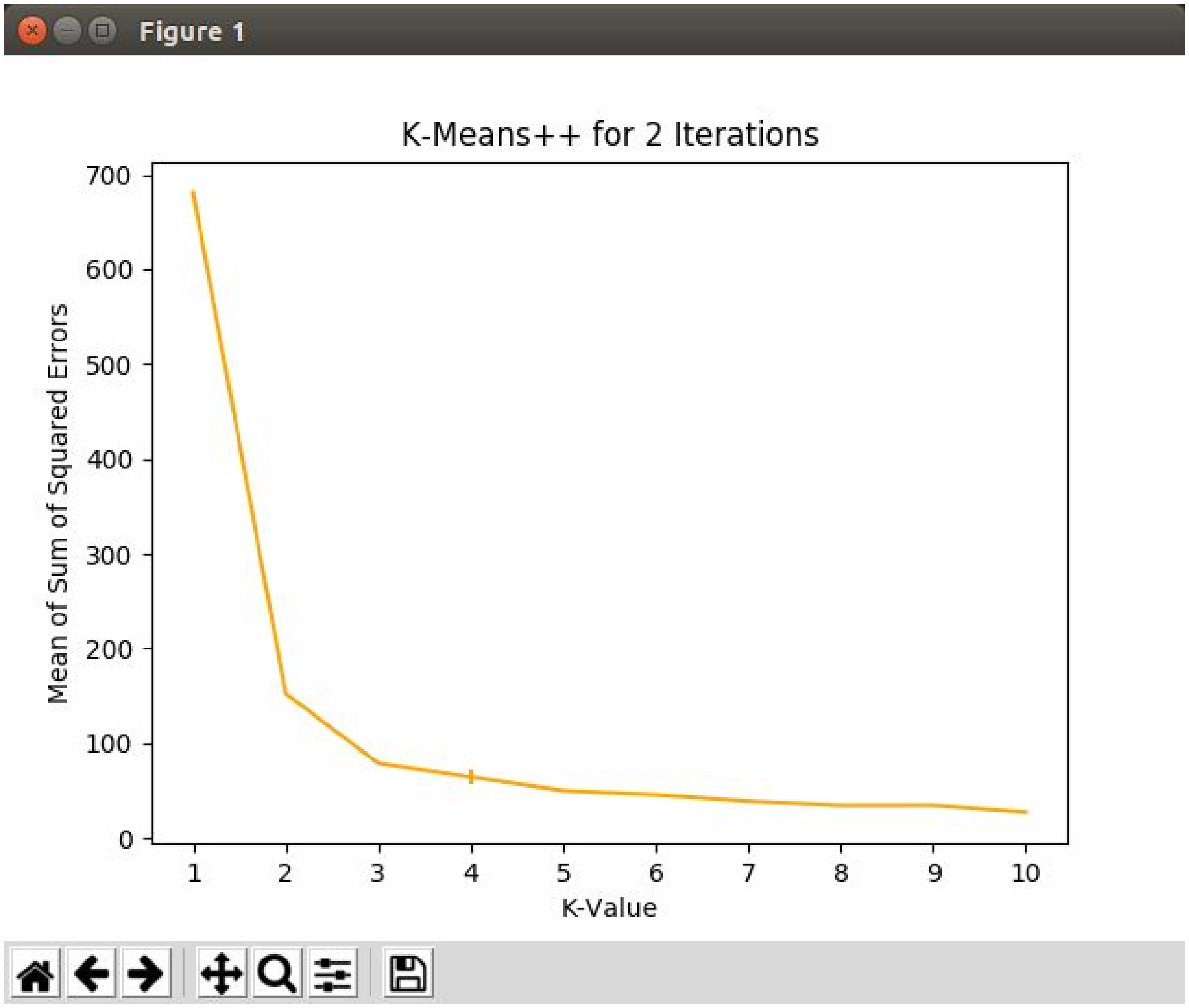


Figure 1
K-Means++ for 2 Iterations
Mean of Sum of Squared Errors
K-Value
0
100
200
300
400
500
600
700
1
2
3
4
5
6
7
8
9
10

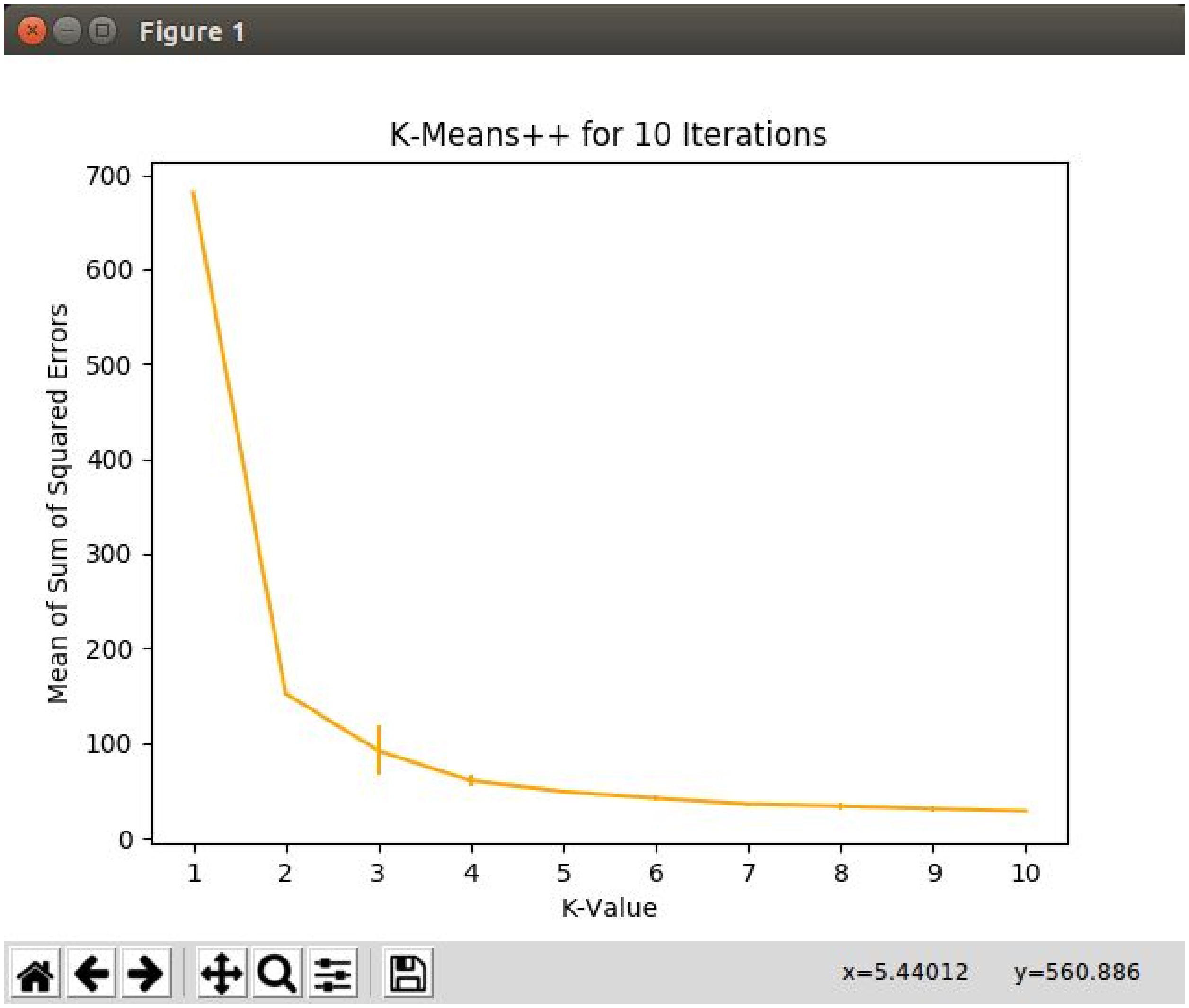
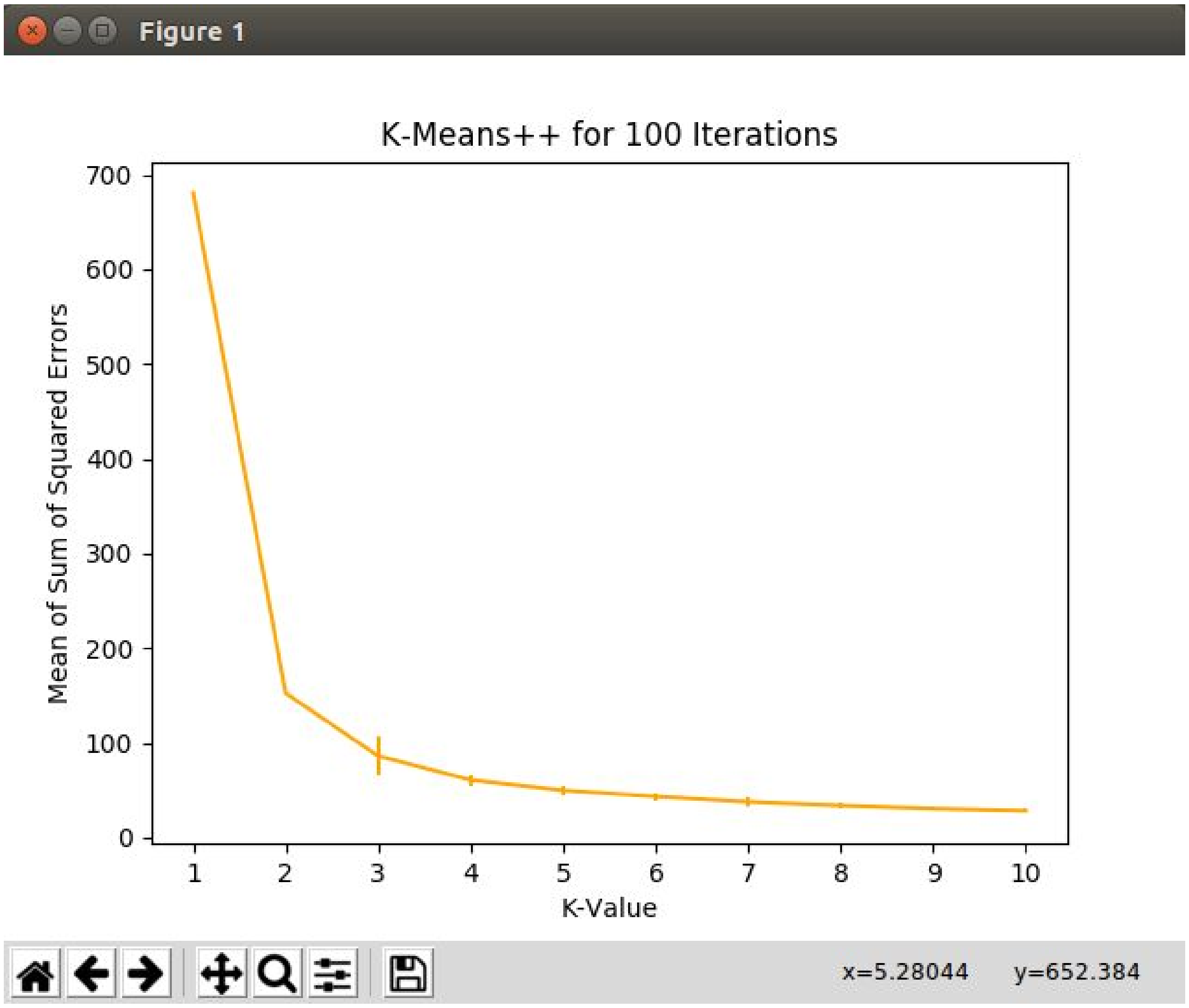


Figure 1
K-Means++ for 10 Iterations
Mean of Sum of Squared Errors
K-Value
0
100
200
300
400
500
600
700
1
2
3
4
5
6
7
8
9
10
x=5.44012 y=560.886



As predicted, the sensitivity graphs for K-Means++ are nearly identical to those of standard K-Means. Once again, this is due to the fact that K-Means++ improves mainly on performance and reliability of the K-Means algorithm, not overall improvement of the clustering.

One interesting difference to note however is that the error bars are relatively smaller for K-Means++. This is due to the added reliability of K-Means++ as it greatly lowers the probability of “unlucky” centroid initialization. The probability of “unluckiness” is still there, but it is significantly smaller than that of standard K-Means.