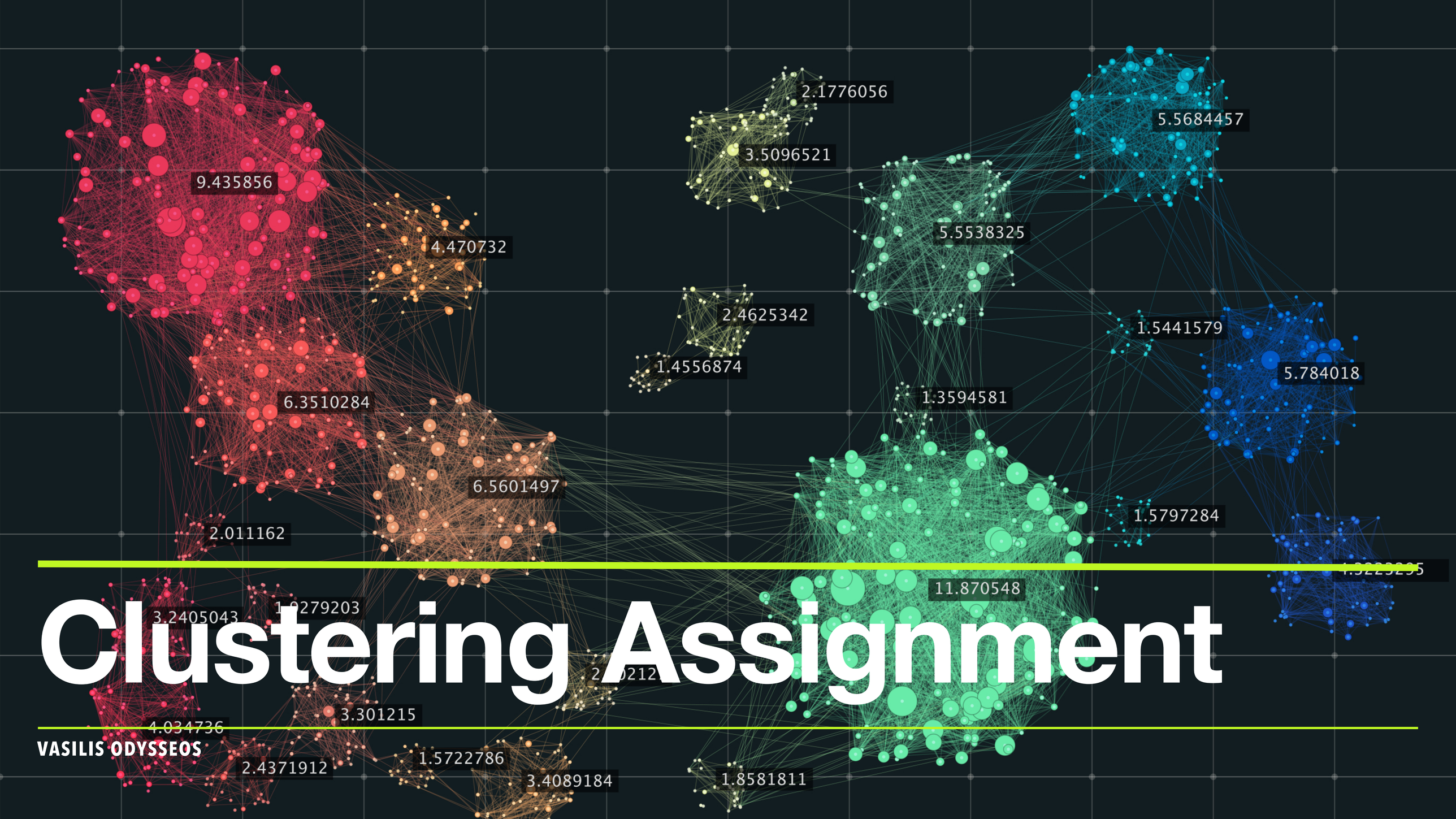
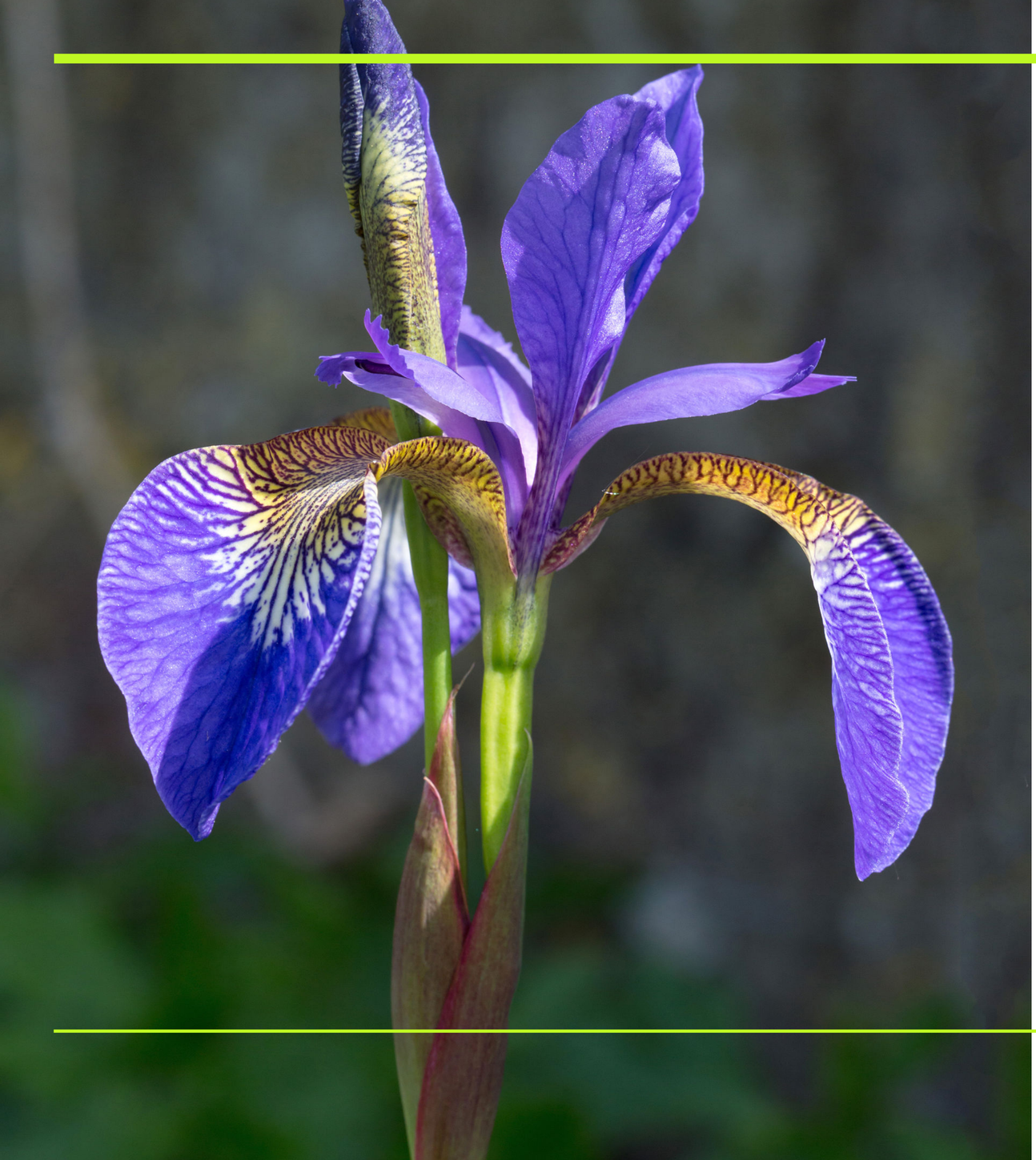


Clustering Assignment

VASILIS ODYSSEOS



Datasets Used

- Iris
  - 150 data points
  - 4 attributes
  - 3 different categories
- Glass
  - 214 data points
  - 9 attributes
  - 6 different categories
- Haberman
  - 306 data points
  - 3 attributes
  - 2 different categories

Linkage Algorithms

```
def single_linkage(filename, k):
   initial_points, clustering = open_file(filename)
    distances = sorted_dist(initial_points)
    while (len(clustering) > k):
        # find the two closest clustering using the sorted
distance list
        for i in range(0, len(clustering)):
            for point in clustering[i]:
                if point[0] == distances[0][0]:
                    cluster1 = i
                if point[0] == distances[0][1]:
                    cluster2 = i
        del(distances[0])
        if (cluster1 == cluster2) or (cluster1 == -1):
            continue
        # merge the two clustering
        for point in clustering[cluster2]:
            clustering[cluster1].append(point)
        del(clustering[cluster2])
    print_clustering(clustering)
    truth = actual_clustering(initial_points)
    hamming_distance = hamming(initial_points,
                               clustering, truth)
    print("Hamming distance from truth", hamming_distance)
    return hamming_distance
```

Linkage Algorithm
Single Linkage

- Gets a list of all edges sorted by distance
- Loops through each edge, merging the clusters to which the two vertices belong
- Does this until k clusters remain

```
def average_linkage(filename, k):
    initial_points, clustering = open_file(filename)
    distances = sq_dist(initial_points)
    while (len(clustering) > k):
        min_avg = 100000000000
        cluster1, cluster2 = -1, -1
        for i in range(0, len(clustering)):
            for m in range(i+1, len(clustering)):
                sum = 0
                for j in range(0, len(clustering[i])):
                    for l in range(0, len(clustering[m])):
                        sum = sum + distances[clustering[i]
                                              [j][0]][clustering[m]
[l][0]]
                avg = sum / (len(clustering[i])*len(clustering[m]))
                if avg < min_avg:
                    min_avg = avg
                    cluster1 = i
                    cluster2 = m
        if (cluster1 == cluster2) or (cluster1 == -1):
            continue
            # merging the clustering
        for point in clustering[cluster2]:
            clustering[cluster1].append(point)
        del(clustering[cluster2])
    print_clustering(clustering)
    truth = actual_clustering(initial_points)
    hamming_distance = hamming(initial_points,
                               clustering, truth)
    print("Hamming distance from truth", hamming_distance)
    return hamming_distance
```

Linkage Algorithm
Average Linkage

- Gets a square matrix of all distances
- Loops through each cluster, finding its average distance from every other cluster
- Merges the two clusters with the smallest average distance
- Does this until k clusters remain

K-Means heuristics

```
def kmeans(initial_points, k, method):
    if method == 'lloyds':
        centroids = lloyds_centroids(initial_points, k)
    if method == 'kmeanspp':
        centroids = kmeanspp_centroids(initial_points, k)
 while True:
        clustering = list()
        for i in range(0, k):
            cluster = list()
            clustering.append(cluster)
        for point in initial_points:
            min_p2c_dist = 100000000000
            min_centr = -1
            for j in range(0, len(centroids)):
                dist = point_dist(point, centroids[j])
                if dist < min_p2c_dist:
                    min_p2c_dist = dist
                    min_centr = j
            clustering[min_centr].append(point)
        new_centroids = find_centroids(initial_points,
clustering)
        if(new_centroids == centroids):
            break
        else:
            centroids = new_centroids
    return clustering, centroids
```

K-Means Heuristic
K-Means Function

- Gets initial centroids depending on which method is used
- Loops through each point in the dataset and assigns it to a cluster, according to its closest centroid
- Finds the centroids of this clustering
- Repeats the process until the new centroids are the same as the old ones
- Returns the clustering and centroids to find k-means cost

K-Means Heuristic
Finding Lloyd's Centroids

```
def lloyds_centroids(points, k):
    # this function selects the initial
centroids for lloyd's method.
    #  It essentially just picks k random
points
    random.seed()

    centroids = list()
    while len(centroids) < k:
        point = random.choice(points)
        new_centroid = point[:]
        new_centroid[0] = len(centroids)
        centroids.append(point)
    centroids.sort
    return centroids
```

- Picks k random points from the dataset as the centroids
- Returns points

```python
def kmeanspp_centroids(points, k):
    # this function selects the intial centroids for Kmeans++.
    # It picks a random point then keeps finding the next points with
probability D(x)^2/(Sum(D(x)^2) for all x),
    # where D(x) is the distance between the point and its nearest
centroid
    centroids = list()
    random.seed()
    point = random.choice(points)
    new_centroid = point[:]
    new_centroid[0] = len(centroids)
    centroids.append(new_centroid)
    while len(centroids) < k:
        distances = list()
        max_dist = 0
        for point in points:
            min_p2c_dist = 10000000
            for centroid in centroids:
                p2c_dist = point_dist(centroid, point)
                if p2c_dist < min_p2c_dist:
                    min_p2c_dist = p2c_dist
            distances.append(min_p2c_dist**2)
        dist_sum = sum(distances)
        for i in range(0, len(distances)):
            distances[i] = distances[i] / dist_sum
            if i > 1:
                distances[i] = distances[i] + distances[i - 1]
        rand = random.random()
        for i in range(0, len(distances)):
            if rand < distances[i]:
                new_centroid = points[i][:]
                new_centroid[0] = len(centroids)
                centroids.append(new_centroid)
                break
    return centroids
```

K-Means Heuristic

Finding K-Means++ Centroids

- Picks a random point from the dataset as the first centroid
- Picks the next point with probability $\frac{D(x)^2}{\sum_{x \in \mathcal{X}} D(x)^2}$
  - D(x) is the distance between x and its closest already selected centroid
- Repeatedly finds the next centroid until k centroids are found
- Returns the centroids

```
def naive_kmeanspp_centroids(points, k):
    centroids = list()
    random.seed()
    point = random.choice(points)
    new_centroid = point[:]
    new_centroid[0] = len(centroids)
    centroids.append(new_centroid)
    while len(centroids) < k:
        max_dist = 0
        for point in points:
            min_p2c_dist = 10000000
            for centroid in centroids:
                p2c_dist = point_dist(centroid, point)
                if p2c_dist < min_p2c_dist:
                    min_p2c_dist = p2c_dist
            if min_p2c_dist > max_dist:
                max_dist = min_p2c_dist
                max_pt = point
        new_centroid = max_pt[:]
        new_centroid[0] = len(centroids)
        centroids.append(new_centroid)
    return centroids
```

K-Means Heuristic

Naive K-Means++ Centroids

- Picks a random point from the dataset as the first centroid
- Assumes that since the point with the largest distance from its nearest centroid has the highest probability of being chosen
- Picks that point
- Repeatedly finds the next centroid until k centroids are found
- Returns the centroids

```
    def run_kmeans(filename, k, method, runs):
        # this function runs the kmans function 100 times
    and selects the clustering with the lowest kmeans cost
        initial_points, clustering = open_file(filename)
        min_cost = 100000000000
        for i in range(runs):
            clustering, centroids = kmeans(initial_points,
    k, method)
            cost = kmeans_cost(clustering, centroids)
            if cost < min_cost:
                min_cost = cost
                best_clustering = clustering
        print_clustering(best_clustering)
        truth = actual_clustering(initial_points)
        hamming_distance = hamming(initial_points,
                                   clustering, truth)
        print("Hamming distance from truth",
    hamming_distance)
        return hamming_distance


def kmeans_cost(clustering, centroids):
    cost = 0
    for i in range(len(clustering)):
        for point in clustering[i]:
            cost = cost + point_dist(point, centroids[i])**2
    return cost
```

K-Means Heuristic

Running n Times To Find Lowest K-Means Cost

- Runs the desired K-means heuristic *run* times
- Returns the clustering with the lowest k-means cost

- Loops between the clusters in the clustering
- Loops between the points in the cluster
- Adds the square of the distance between the point and the centroid of its cluster
- Returns the cost

```
def hamming(initial_points, c1, c2):
    h1 = 0
    h2 = 0
    for i in range(0, len(initial_points)):
        for j in range(i+1, len(initial_points)):
            cluster1 = -1
            cluster2 = -1
            cluster3 = -1
            cluster4 = -1
            for k in range(0, len(c1)):
                for point in c1[k]:
                    if point[0] == initial_points[i][0]:
                        cluster1 = k
                    if point[0] == initial_points[j][0]:
                        cluster2 = k
                    if(cluster1 != -1 and cluster2 != -1):
                        break
            for k in range(0, len(c2)):
                for point in c2[k]:
                    if point[0] == initial_points[i][0]:
                        cluster3 = k
                    if point[0] == initial_points[j][0]:
                        cluster4 = k
                    if(cluster3 != -1 and cluster4 != -1):
                        break
            if(cluster1 == -1 or cluster2 == -1 or cluster3 ==
-1 or cluster4 == -1):
                if(cluster1 == cluster2 and cluster3 != cluster4):
                    h1 = h1 + 1
                if(cluster1 != cluster2 and cluster3 == cluster4):
                    h2 = h2 + 1
    return (h1+h2) / math.comb(len(initial_points), 2)
```

Hamming Distance

Finding How Truthful the Clusterings Are

- Loops through each pair of points in the data set
- Finds which cluster each point is in both clusterings
- Checks if both edges are in-cluster or both out-of cluster
- Keeps track of how many inconsistencies there are

Results

Dataset	Single Linkage	Average Linkage	Lloyds n=1	Lloyds n=100	Kmeans++ n=1	Kmeans++ n=100	naive Kmeans++ n=1	naive Kmeans++ n=100
Iris	0.2234	0.1077	0.1263	0.1263	0.1203	0.1203	0.1203	0.1203
Glass	0.7030	0.6704	0.2954	0.2798	0.3852	0.3887	0.3892	0.4069
Haberman	0.3874	0.3905	0.5009	0.5009	0.5009	0.5016	0.3679	0.3679