Data Science Lab

Lab 4 solution

0.1 2D Gaussian clusters

0.1.1 Exercise 1.1

To load the dataset, the `numpy.loadtxt` method can be used. This time, the dataset file contains an header in the first line. We will skip it using the method iself.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

[2]:
```
X_gauss = np.loadtxt("2D_gauss_clusters.txt", delimiter=",", skiprows=1)
X_gauss
```

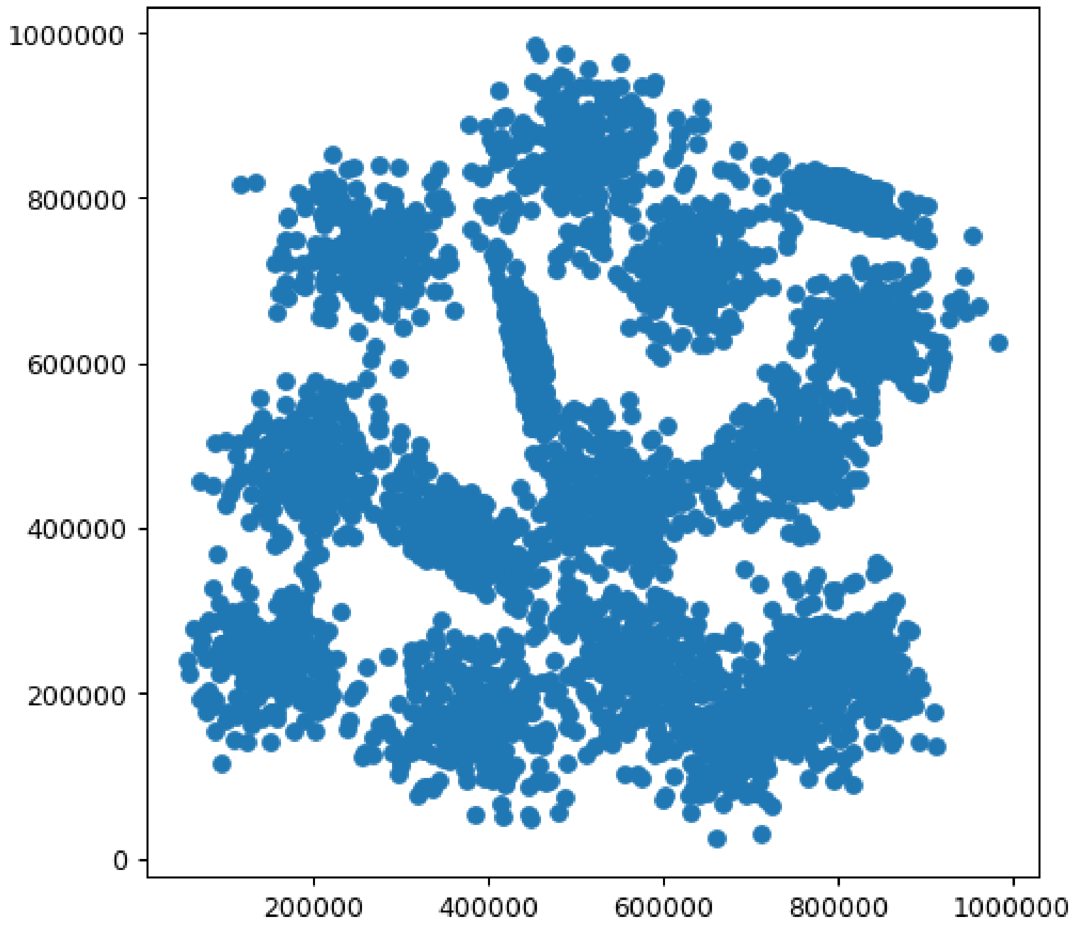
[2]:
```
array([[845753., 636607.],
       [812954., 643720.],
       [868217., 609046.],
       ...,
       [706472., 171207.],
       [659128., 142611.],
       [665898.,  98088.]])
```

0.1.2 Exercise 1.2

Let's use now `matplotlib` to explore our 2D data. Since we are likely going to need a scatter plot multiple times, we can define a simple function to handle it.

[3]:
```
def plot_2d_scatter(X):
    """Display a 2D scatter plot

    :param X: input data points, array
    :return: fig, ax, objects
    """
    fig, ax = plt.subplots(figsize=(6, 6), dpi=90)
    ax.scatter(X[:,0], X[:,1])
    return fig, ax # use them for further modifications
```

[4]:
```
_, _ = plot_2d_scatter(X_gauss) # the two underscores let you discard the␣
 ↪returned values
```



Gaussian, globular clusters are easily visible in the euclidean space. At first sight, you can count up to 15 clusters. Most of them have a regular shape, while some others are more elongated. Given this distribution, we known from theory that the K-means algorithm can achieve positive clustering performance (provided that the right number of cluster is chosen).

0.1.3 Exercises 1.3 and 1.4

Let's implement now our version of the K-means algorithm. We will use a custom Python class. The basic blocks will be: * a `fit_predict` method exposed to the users of the class * a stateful structure: we do want to save the final clusters labels and centroids * some internal plotting functions * a `dump_to_file` method to save cluster labels to CSV file (see Exercise 1.4)

[5]:
```
class KMeans:

    def __init__(self, n_clusters, max_iter=100):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.centroids = None
        self.labels = None
```

```python
    def __plot_clusters(self, X, labels, c=None):
        fig, ax = plt.subplots(figsize=(6,6), dpi=90)
        ax.scatter(X[:,0], X[:,1], c=labels, cmap='Set3')
        if c is not None:
            ax.scatter(c[:,0], c[:,1], marker="*", color="red")

        return fig, ax

    def __init_random_centroids(self, X):
        c_idx = np.random.randint(0, X.shape[0], self.n_clusters)
        c = X[c_idx] # fancy-indexing
        return c

    def fit_predict(self, X, plot_clusters=False, plot_steps=5):
        """Run the K-means clustering on X.

        :param X: input data points, array, shape = (N,C).
        :return: labels : array, shape = N.
        """
        # For each point, store the positional index of the closest centroid
        nearest = np.empty(X.shape[0])

        c_out = self.__init_random_centroids(X)
        for j in range(self.max_iter):
            nearest_prev = nearest

            # For each point in X, compute the squared distance from each␣
↪centroid
            # dist will have shape: (n_clusters, X.shape[0])
            dist = np.array([np.sum((X-c_out[i])**2, axis=1) for i in␣
↪range(self.n_clusters)])

            # Find the nearest centroid for each point using argmin
            # nearest will have shape: (X.shape[0],)
            nearest = dist.argmin(axis=0)

            if plot_clusters and (j % plot_steps == 0):
                fig, ax = self.__plot_clusters(X, nearest, c_out)
                ax.set_title(f"Iteration {j}")

            # Early stopping if centroids have not changed
            if np.array_equal(nearest, nearest_prev):
                print(f"Early stopping at iteration {j}!")
                break
            # For each cluster, compute the average coordinates considering␣
↪only points currently
            # assigned to it. Then, use them as the new centroid coordinates.
            for i in range(self.n_clusters):
                c_temp = X[nearest == i].mean(axis=0) # indexing with a mask
                if not np.isnan(c_temp).any(): # handle the case of an empty␣
↪cluster
                    c_out[i] = c_temp

        self.centroids = c_out
        self.labels = nearest
        return self.labels

    def dump_to_file(self, filename):
        """Dump the evaluated labels to a CSV file."""
        with open(filename, 'w') as fp:
            fp.write('Id,ClusterId\n')
            for i, label in enumerate(self.labels):
                fp.write(f'{i},{label:d}\n')
```

There are several things worth noting in the code above: 1. since data are not shuffled during the execution, the class makes use of positional indices. The `nearest` array is a good example of that. For each point, it keeps track of the index of the centroid of the cluster that point is assigned to. In other words, if the *ith* item of `nearest` has value *j*, we know that the *ith* point of the dataset is assigned to the *jth* cluster. Using this convention, the choice of the cluster Ids is straightforward: the *ith* cluster will have ID = *i* 2. this is a typical case in which NumPy can significantly speed up the execution. NumPy's broadcasting feature is used to compute the squared distances from each cluster centroid. Then, `argmin` is used to soft-sort them and find the closest centroid Id. Finally, `array_equal` lets you efficiently compare the current and previous centroid positions (if we wanted to compare them with a tolerance, we could have used the `numpy.allclose` method) 3. some internal methods have a double underscore as name prefix. As you should know, Python does not provide the concept of private attributes or methods by choice. However, programmers that define members intended to be private can make use of name mangling (it remains a design choice, it does not prevent other to access to those members). Name mangling is commonplace in many languages. In Python, it is enabled by adding two leading underscore to your member name (actually, you are also restricted to have at most one trailing underscore). You can read more on name mangling here.

We strongly suggest you to stop and try to fully understand the first two points. For your convenience, we commented the shape of the numpy arrays involved at their respective lines.

Let's try now to use our class and save our labels to file.

[6]:
```
np.random.seed(0)
kmeans_model = KMeans(10)
l_gauss = kmeans_model.fit_predict(X_gauss)
l_gauss
```

```
Early stopping at iteration 11!
```

```
/Users/giuseppe/miniconda3/envs/atsa-py37/lib/python3.7/site-
```

```
packages/numpy/core/numeric.py:2591: RuntimeWarning: invalid value encountered
in equal
  return bool(asarray(a1 == a2).all())
```

[6]:
```
array([1, 1, 1, ..., 5, 5, 5])
```

The algorithm has stopped after 11 iterations since every centroid has not moved between iteration 10 and 11. In other words, at the 11th iteration, no point has changed its closest centroid (i.e. its membership cluster).

Note also line 1 of the previous cell: it initializes the NumPy's random generator with a specific seed. With a fixed random generator, the same points will be chosen as initial centroids among different runs. This is commonplace whenever you want your results to be reproducible. Many scikit-learn classes offer you this functionality via the `random_state` parameter.

[7]:
```
kmeans_model.dump_to_file('gauss_labels.csv')
```

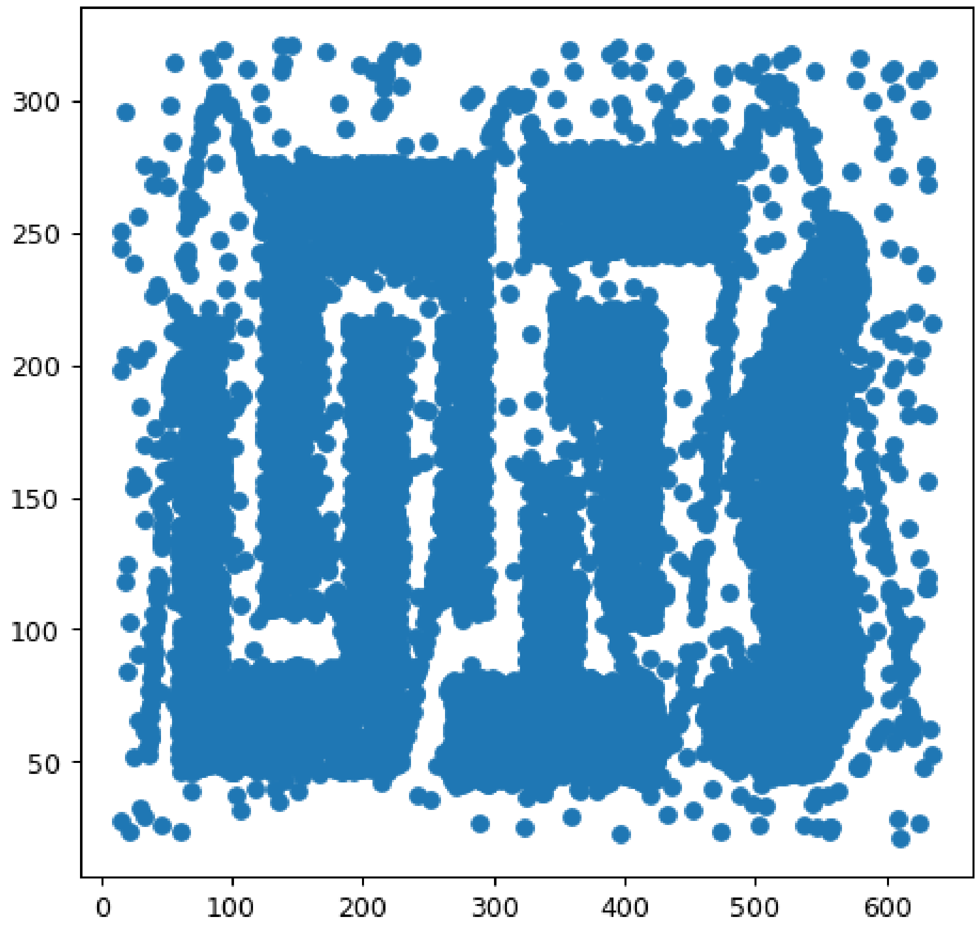
0.1.4 Exercise 1.5

Let's analyze now the Chameleon dataset just as before.

[8]:
```
X_cham = np.loadtxt("chameleon_clusters.txt", delimiter=",", skiprows=1)
X_cham
```

[8]:
```
array([[ 68.601997, 102.491997],
       [454.665985, 264.80899 ],
       [101.283997, 169.285995],
       ...,
       [267.605011, 141.725006],
       [238.358002, 252.729996],
       [159.242004, 177.431   ]])
```

[9]:
```
_, _ = plot_2d_scatter(X_cham)
```



As you can notice, there are six interleaved areas with an high density of points and irregular shape. Also, the dataset looks much more noisier (note the sine wave superimposed). Given this distribution, we known from theory that the K-Means algorithm might suffer even choosing the right number of clusters.

0.1.5 Exercise 1.6

It is time to enhance our plotting toolkit. We hence define a new function that highlights the final centroids. Specifically, we can use a different marker style and color for them.

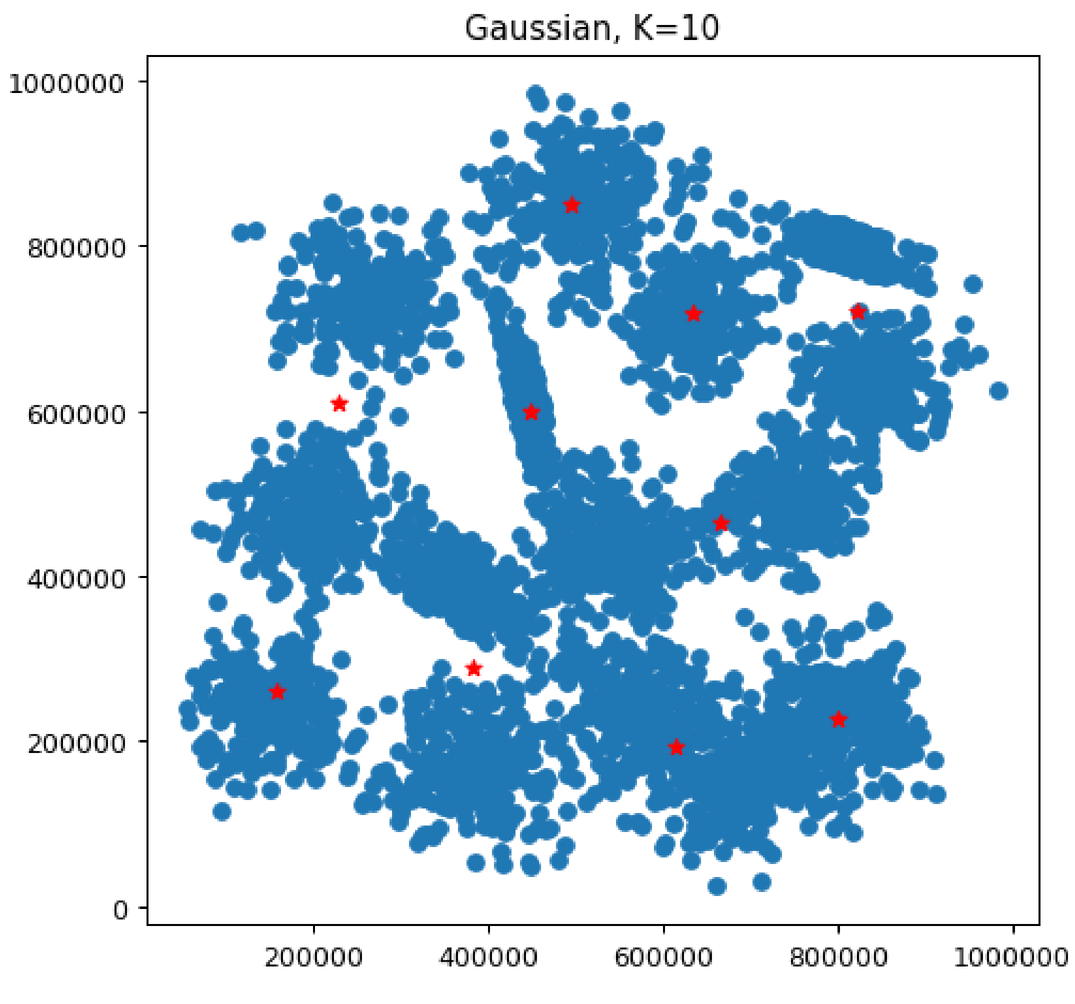
[10]:
```python
def plot_centroids(X, c, title=None):
    fig, ax = plot_2d_scatter(X)
    ax.scatter(c[:,0], c[:,1], marker="*", color="red")

    if title:
        ax.set_title(title)
    return fig, ax
```

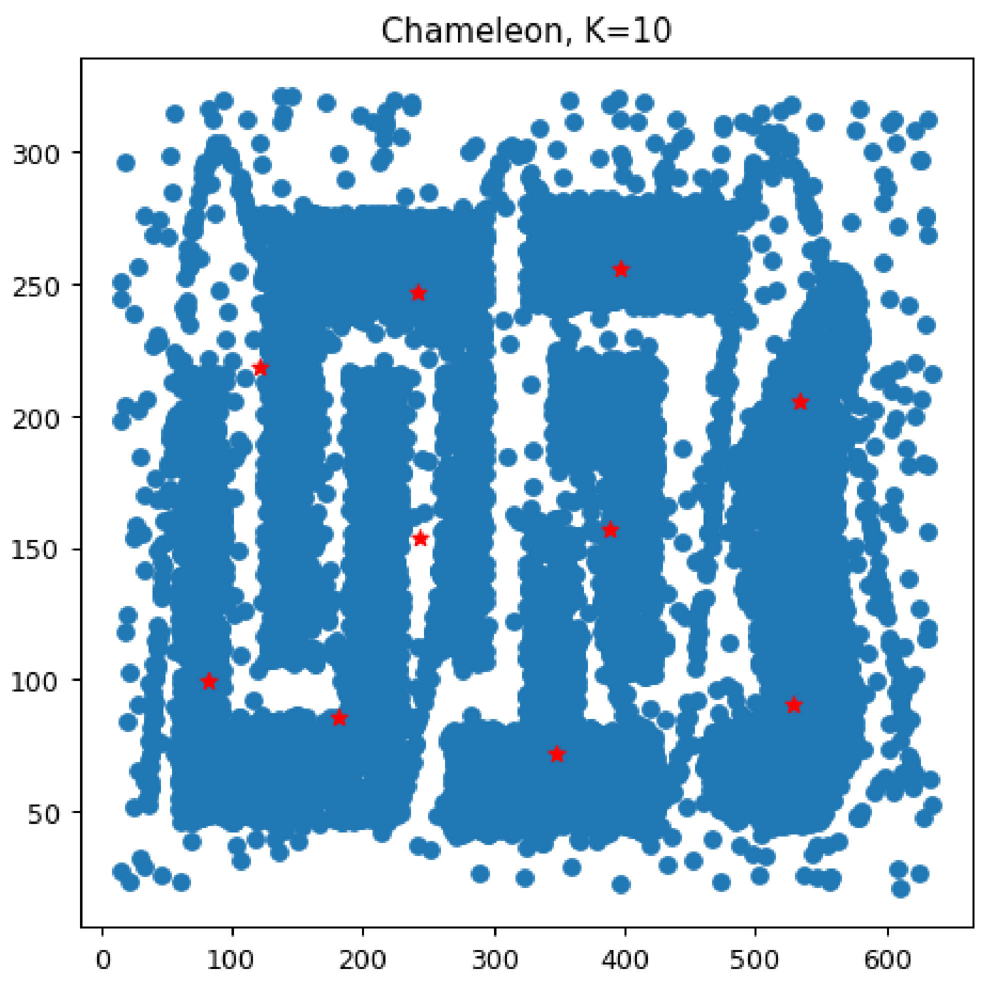
Let's now try to inspect the final centroids obtained in both our datasets.

[11]:
```
np.random.seed(0)
k_gauss = KMeans(10)
k_cham = KMeans(10)

k_gauss.fit_predict(X_gauss)
k_cham.fit_predict(X_cham)

plot_centroids(X_gauss, k_gauss.centroids, title="Gaussian, K=10")
plot_centroids(X_cham, k_cham.centroids, title="Chameleon, K=10")
```

```
Early stopping at iteration 11!
Early stopping at iteration 29!
```

[11]:
```
(<Figure size 540x540 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x109b78358>)
```



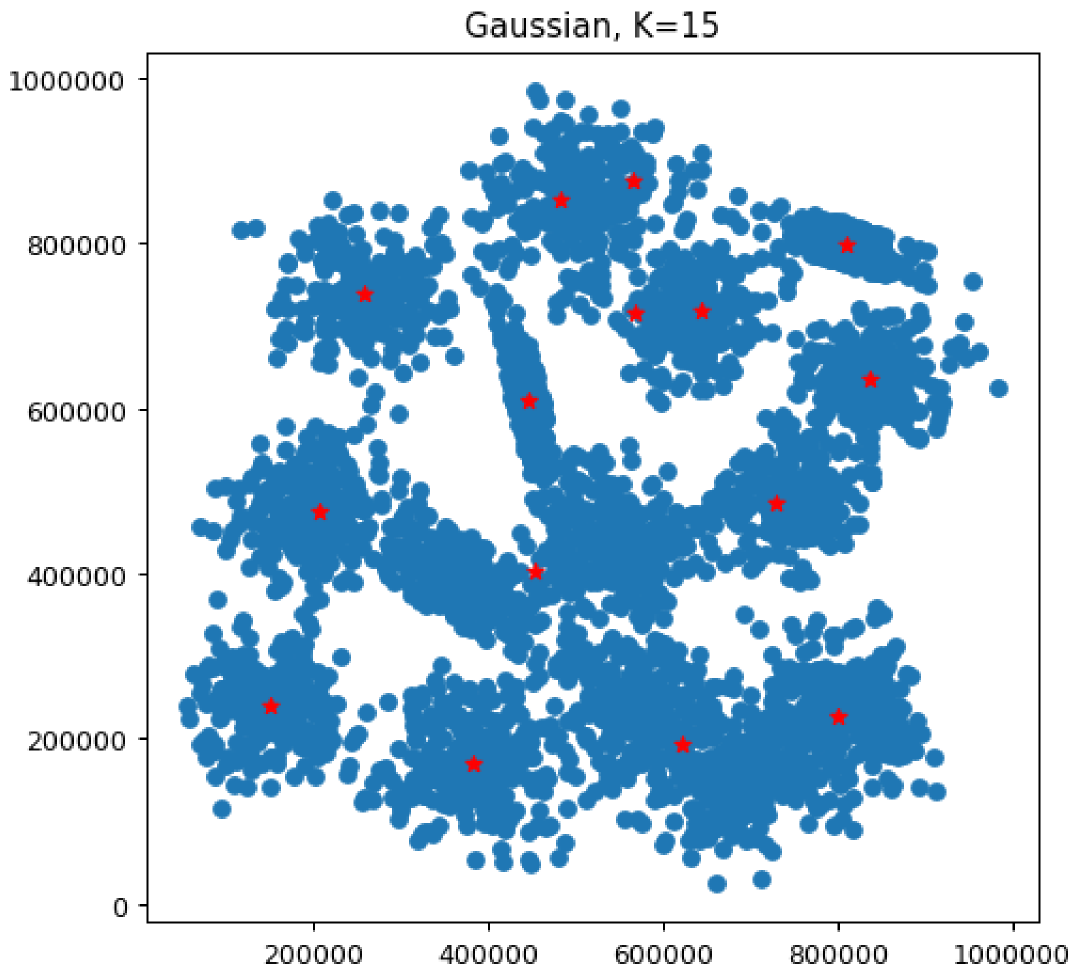


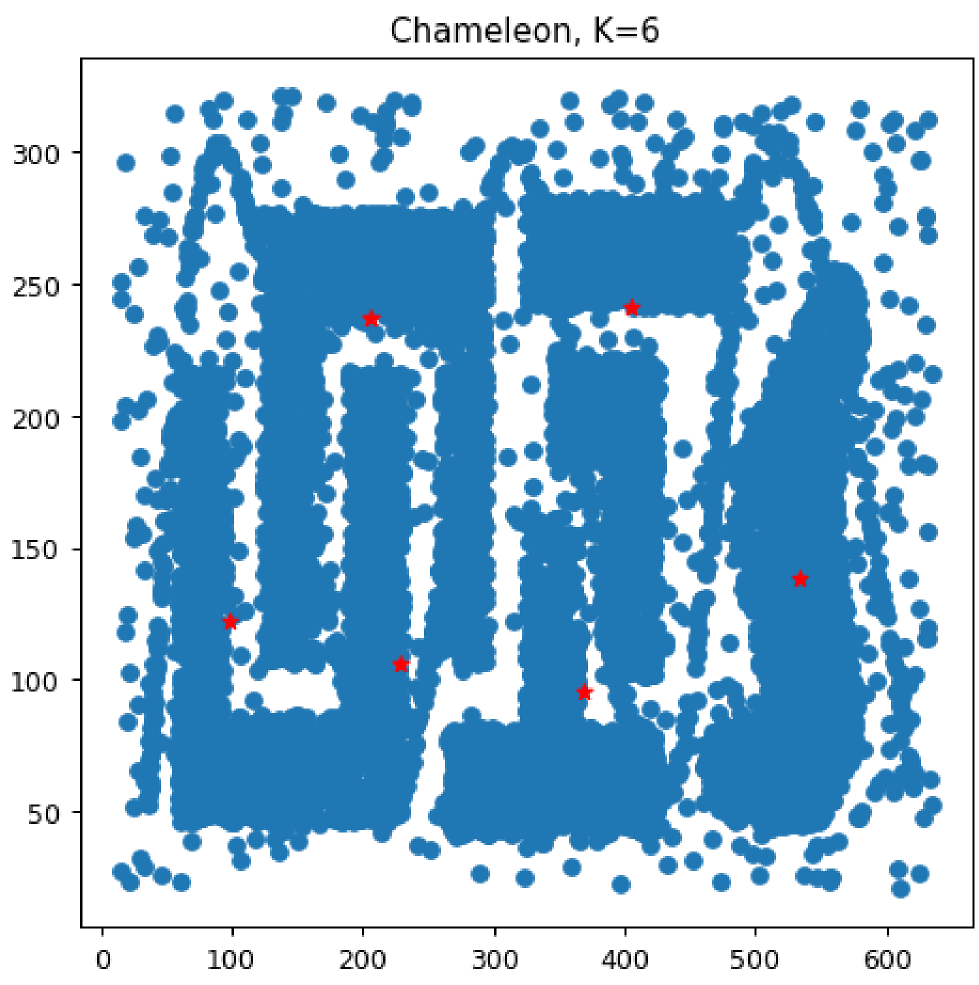
We can visually see that the cluster identification might be improved choosing a different K. Our initial choice is too low for Gaussian data while too high for Chameleon.

[12]:
```
np.random.seed(0)
k_gauss = KMeans(15)
k_cham = KMeans(6)

k_gauss.fit_predict(X_gauss)
k_cham.fit_predict(X_cham)

plot_centroids(X_gauss, k_gauss.centroids, title="Gaussian, K=15")
plot_centroids(X_cham, k_cham.centroids, title="Chameleon, K=6")
```

```
Early stopping at iteration 40!
Early stopping at iteration 24!
```

[12]:
```
(<Figure size 540x540 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x109a24f98>)
```


Gaussian, K=15
1000000
800000
600000
400000
200000
0
200000
400000
600000
800000
1000000



Here some interesting happened. On one hand, the Gaussian dataset was clustered in a better way (again, visually speaking). Yet, there are two pairs of centroids in which the two points are close enough to stay in the same globular cluster. This is likely due to the random initialization of centroids (in literature, there exist other approaches rather than randomly picking them). On the other hand, centroids identified in the Chameleon dataset did not match the real data distribution. This is exactly what we expected: the initial distribution has shape and sizes the K-means algorithm can hardly fit.

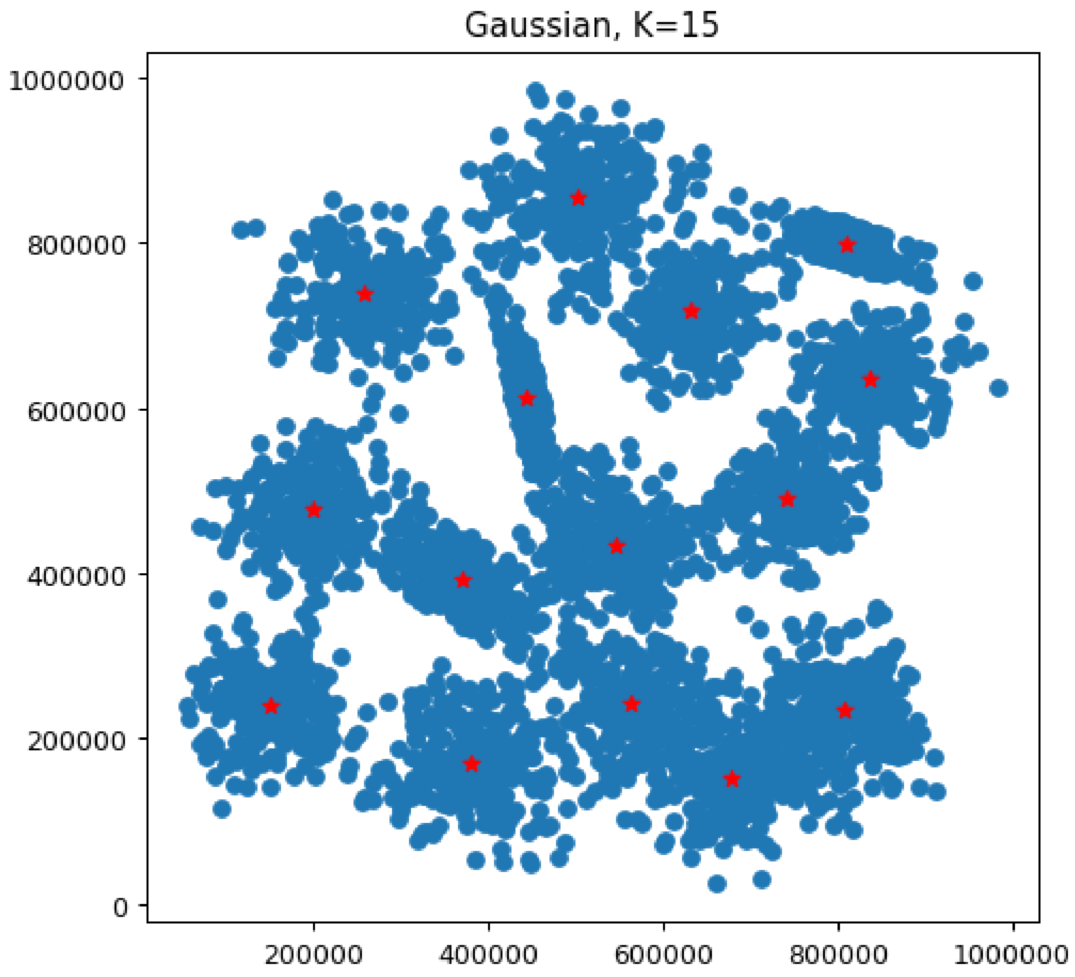
For the sake of completeness, you can run multiple executions with different seed generators. Results are likely to remain the same for Chameleon clusters. Instead, a different random initialization, for the first dataset, could lead to a better result, as shown below.

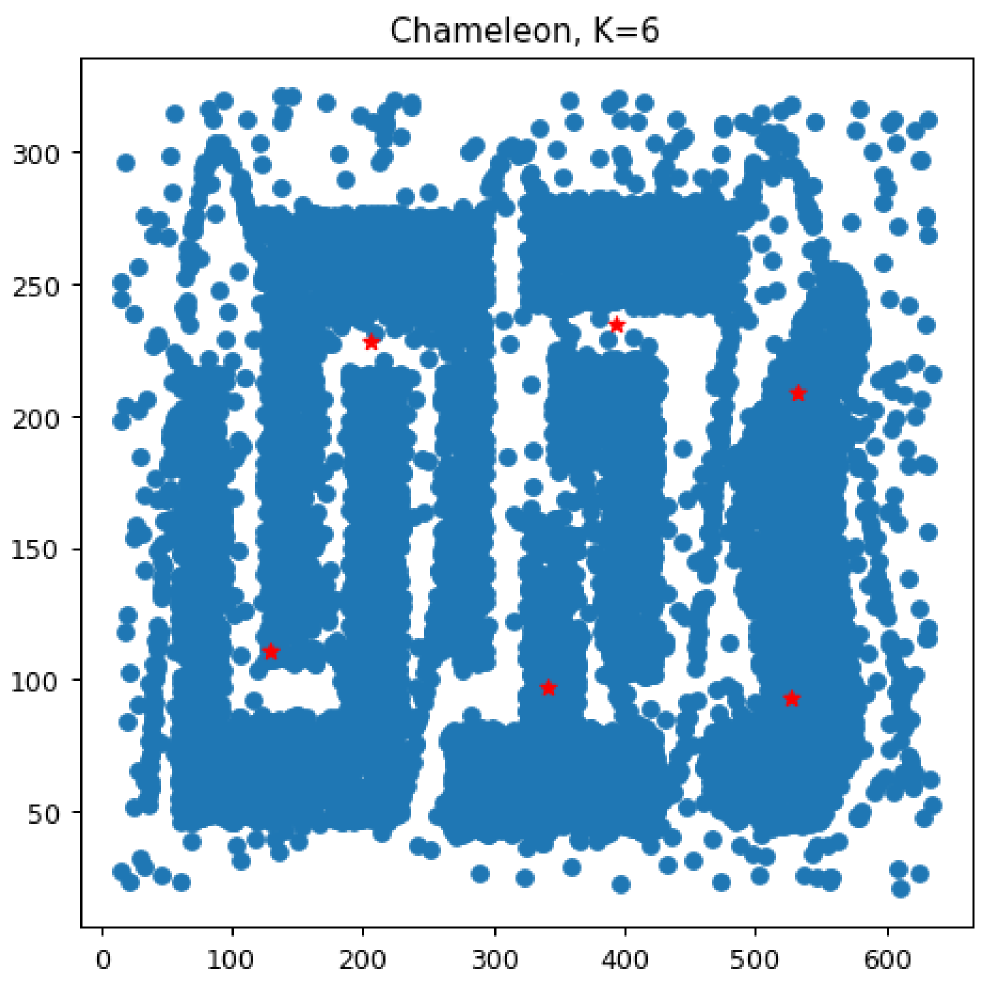
[13]:
```
np.random.seed(6)
k_gauss = KMeans(15)
k_cham = KMeans(6)

k_gauss.fit_predict(X_gauss)
k_cham.fit_predict(X_cham)
```

```
plot_centroids(X_gauss, k_gauss.centroids, title="Gaussian, K=15")
plot_centroids(X_cham, k_cham.centroids, title="Chameleon, K=6")
```

```
Early stopping at iteration 14!
Early stopping at iteration 15!
```

[13]:
```
(<Figure size 540x540 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x109a0bdd8>)
```





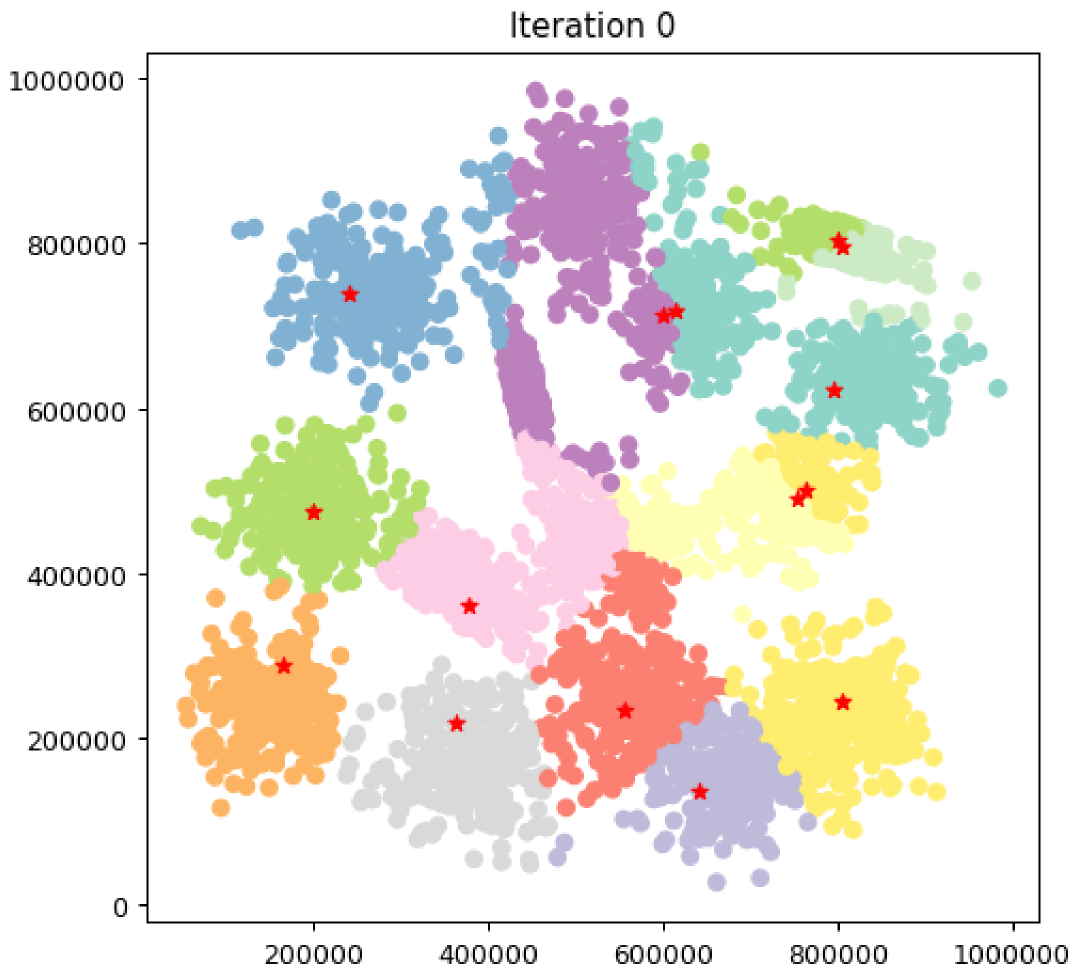
0.1.6 Exercise 1.7

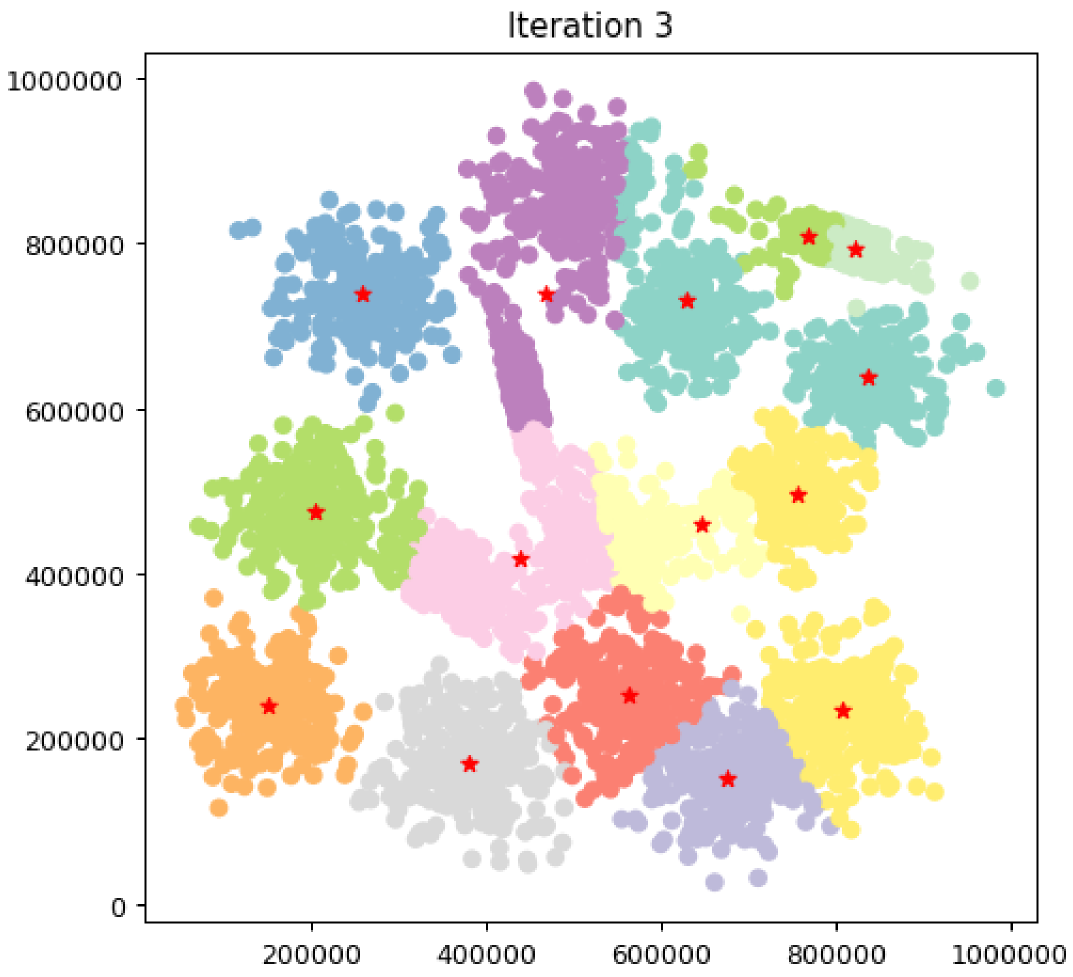
We might be interested to the intermediate position of centroids, other than the final ones. We included this plotting functionality in the `KMeans` class defined above via the `plot_clusters` and `plot_steps` parameters of `fit_predict`. The plot is created in `__plot_clusters`. Take a close look at the `c` and `cmap` parameters. `c` accepts a list used to specify which points must have the same color. Points at indices with equal values in that list will have the same color. `cmap` instead lets you choose a color map among the ones predefined.

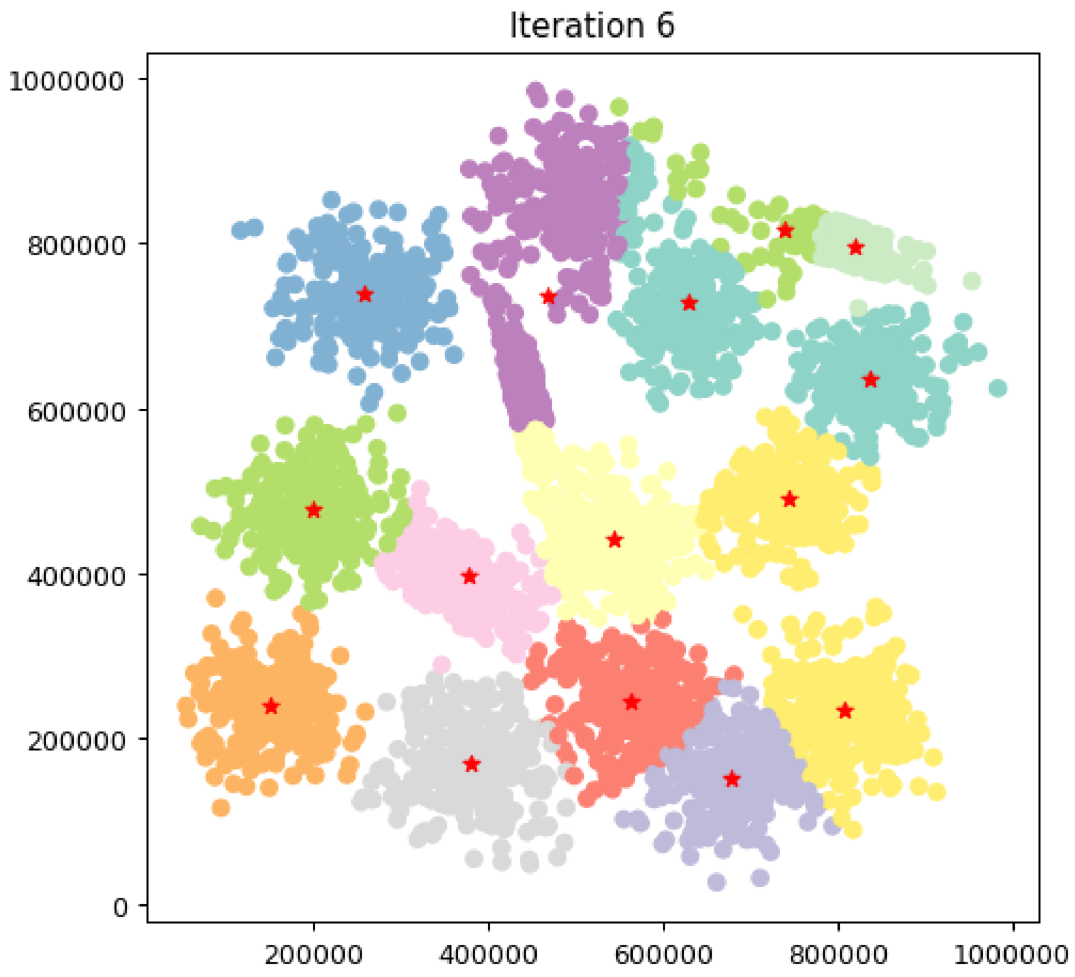
Let's test it with Gaussian clusters.

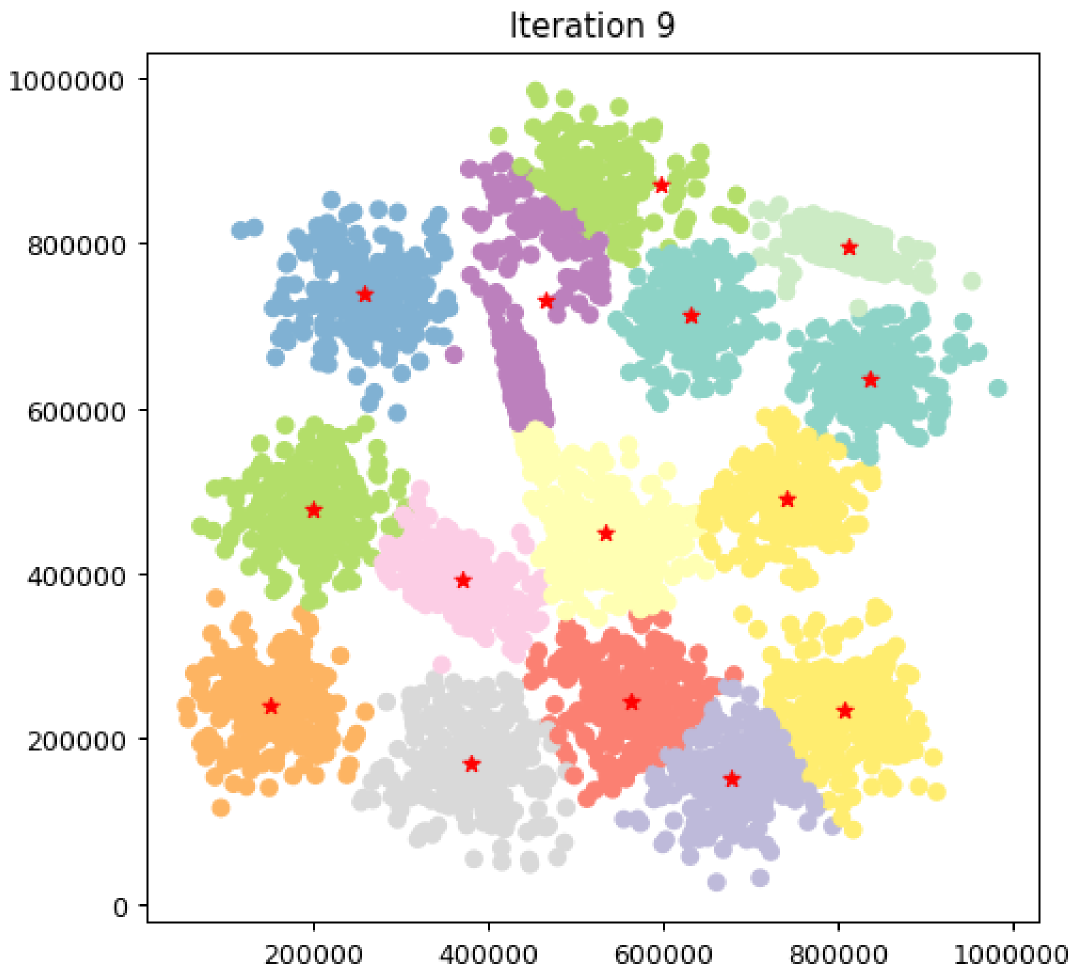
[14]:
```
np.random.seed(6)
k_gauss = KMeans(15)
_ = k_gauss.fit_predict(X_gauss, True, 3)
```

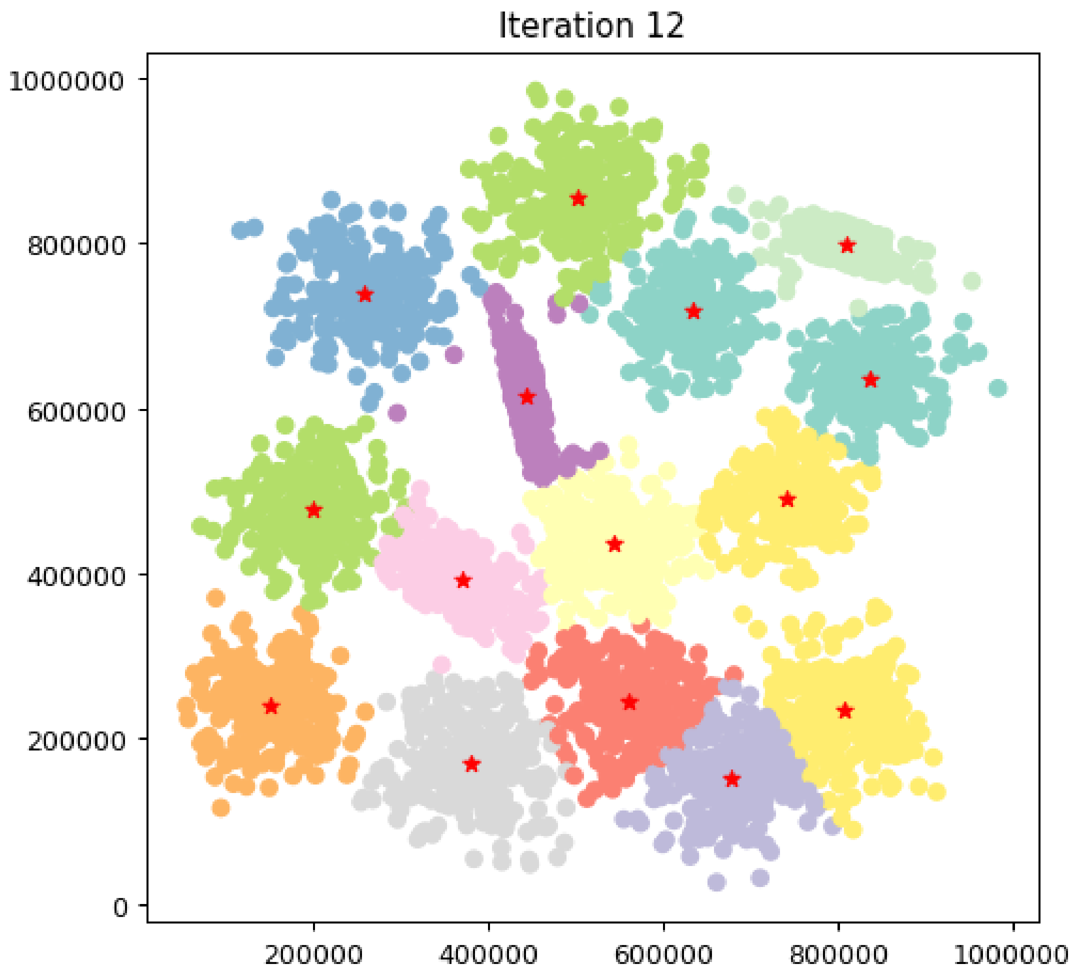
```
Early stopping at iteration 14!
```


Iteration 0
1000000
800000
600000
400000
200000
0
200000
400000
600000
800000
1000000


Iteration 3
1000000
800000
600000
400000
200000
0
200000
400000
600000
800000
1000000


Iteration 6
1000000
800000
600000
400000
200000
0
200000
400000
600000
800000
1000000


Iteration 9
1000000
800000
600000
400000
200000
0
200000
400000
600000
800000
1000000



0.1.7 Exercise 2.2

So far, we have talked about "good" and "bad" clustering results in relation with the capability of K-means to adapt to different data distribution. Our discussions were mainly founded on the final centroids positions. This *visual* approach could not be feasible with many more points and dimensions.

Whenever numerical data is processed, several distance-based metrics can be used to assess the quality of our clustering. Let's now focus on the Silhouette measure. We can define a couple of functions to compute its value (namely `compute_a` and `compute_b`). Note that we will use the euclidean distance.

[15]:
```python
def compute_a(x, x_label, X, labels):
    c_idx = np.where(labels == x_label) # we are including the point itself...
    a = np.mean(np.linalg.norm(x - X[c_idx], axis=1)) # ...because this␣
↪distance will be 0 when computed from itself.
    return a
```

```
def compute_b(x, x_label, X, labels):
    bs = []
    other_labels = np.unique(labels[labels != x_label]) # array with all␣
 ↪cluster Ids different from the current one

    for ol in other_labels:
        c_idx = np.where(labels == ol)
        b = np.mean(np.linalg.norm(x - X[c_idx], axis=1))
        bs.append((ol, b))

    neigh, b = min(bs, key=lambda x: x[1])
    return neigh, b

def silhouette_samples(X, labels):
    """Evaluate the silhouette for each point and return them as a list.

    :param X: input data points, array, shape = (N,C).
    :param labels: the list of cluster labels, shape = N. :return: silhouette :␣
 ↪array, shape = N
    """
    silhouette = np.zeros(X.shape[0])
    for idx in range(X.shape[0]):
        x = X[idx]
        x_label = labels[idx]

        # compute a
        a = compute_a(x, x_label, X, labels)
        # compute b
        _, b = compute_b(x, x_label, X, labels)

        silhouette[idx] = (b - a) / np.max([a, b])

    return silhouette

def silhouette_score(X, labels):
    """Evaluate the silhouette for each point and return the mean.

    :param X: input data points, array, shape = (N,C).
    :param labels: the list of cluster labels, shape = N. :return: silhouette :␣
 ↪float
    """
    return np.mean(silhouette_samples(X, labels))
```

Let's see how our clustering performs.

[16]:
```
np.random.seed(6)
k_gauss = KMeans(15)
k_cham = KMeans(6)
```

```
l_gauss = k_gauss.fit_predict(X_gauss)
l_cham = k_cham.fit_predict(X_cham)
sil_gauss = silhouette_score(X_gauss, l_gauss)
sil_cham = silhouette_score(X_cham, l_cham)

print('Gaussian clusters, average silhouette:', sil_gauss)
print('Chameleon clusters, average silhouette:', sil_cham)
```

```
Early stopping at iteration 14!
Early stopping at iteration 15!
Gaussian clusters, average silhouette: 0.6272254541628367
Chameleon clusters, average silhouette: 0.4134950882250952
```

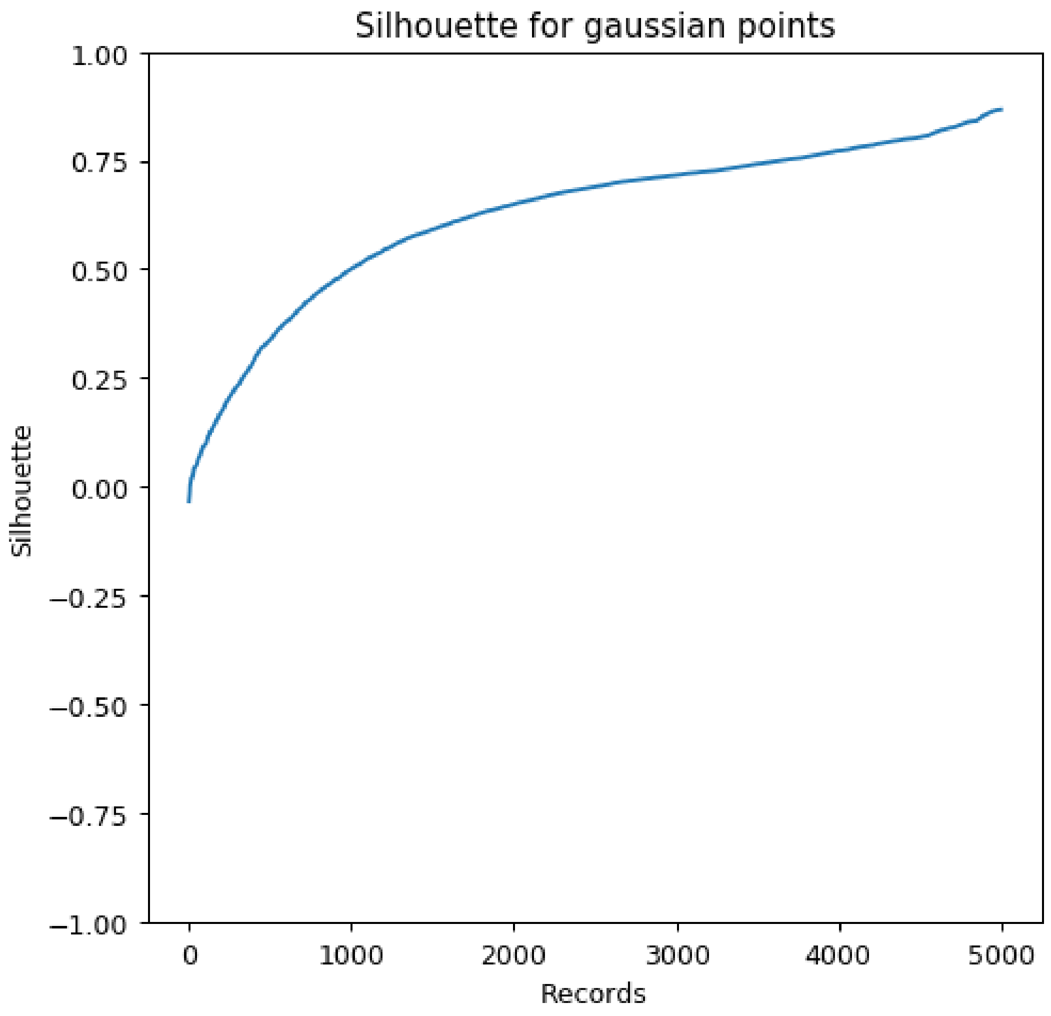
As expected from our initial inspection, we achieved a much higher silhouette score on the first dataset.

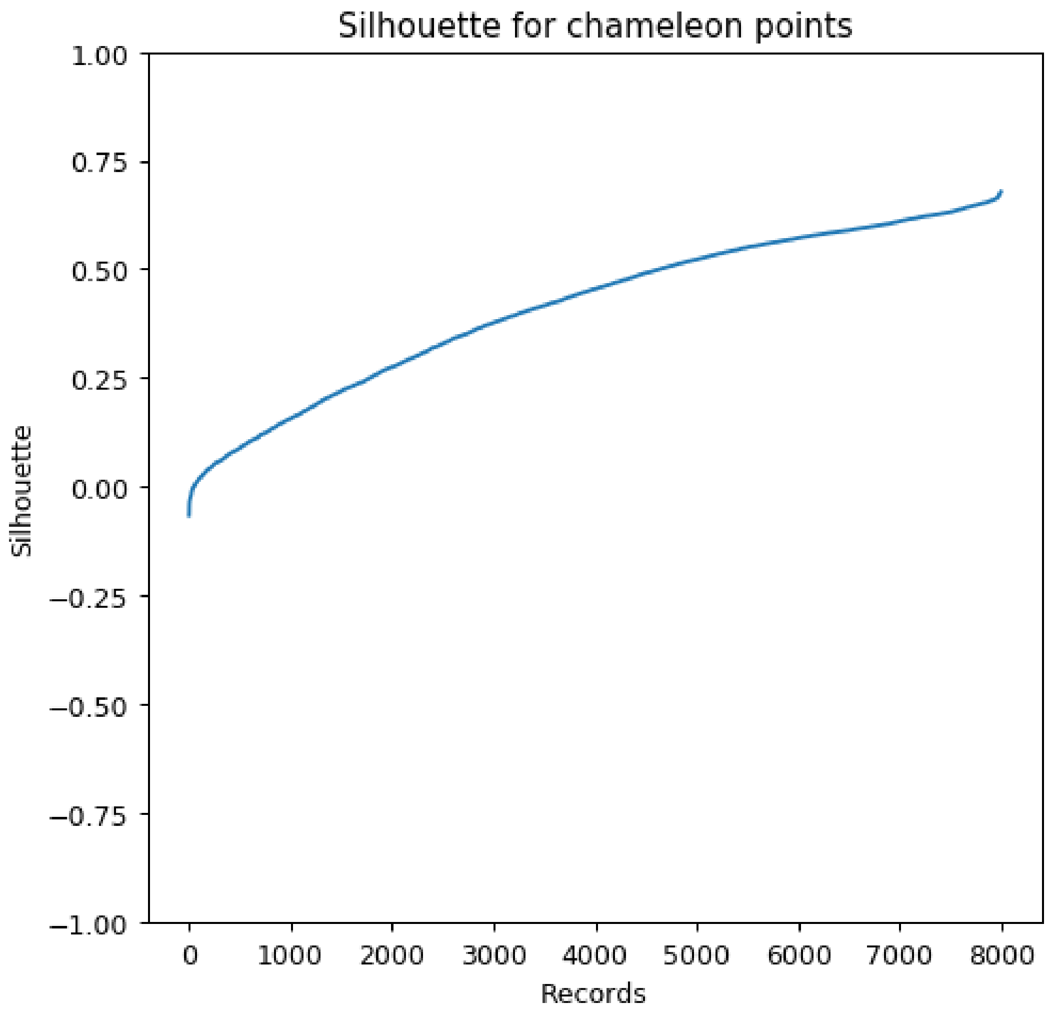
0.1.8 Exercise 2.2

Let's now analyze how the Silhouette values are distributed between our points. To do so, we can plot them in ascending order on the x-axis.

[17]:
```
def plot_silhoeutte(silhouette, title=None):
    fig, ax = plt.subplots(figsize=(6,6), dpi=90)
    ax.plot(np.sort(silhouette))
    ax.set_ylabel("Silhouette")
    ax.set_ylim(-1,1)
    ax.set_xlabel("Records")
    if title:
        ax.set_title(title)
    return fig, ax
```

[18]:
```
_, _ = plot_silhoeutte(silhouette_samples(X_gauss, l_gauss), "Silhouette for␣
 ↪gaussian points")
_, _ = plot_silhoeutte(silhouette_samples(X_cham, l_cham), "Silhouette for␣
 ↪chameleon points")
```


Silhouette for gaussian points
Silhouette
Records
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1000
2000
3000
4000
5000



Again, as expected, we got more points shifted towards the maximum silhouette value (i.e. 1) clustering the first dataset rather than the second one.

0.1.9 Exercise 2.3

Until now, we analized results achieved with the right number of cluster as K. Even with the best educational purposes, there is a trick here. We did that because we knew approximately the number of clusters thanks to a prior visual inspection. In many real-world tasks, your data can be spanned on more than two or three dimensions and you cannot visualize them. Hence, it is commonplace to search for the K value that leads to the best possible clustering subdivision. For what concerns the silhouette measure, the higher is the average silhouette the better are the intra-cluster cohesion and the inter-cluster separation.

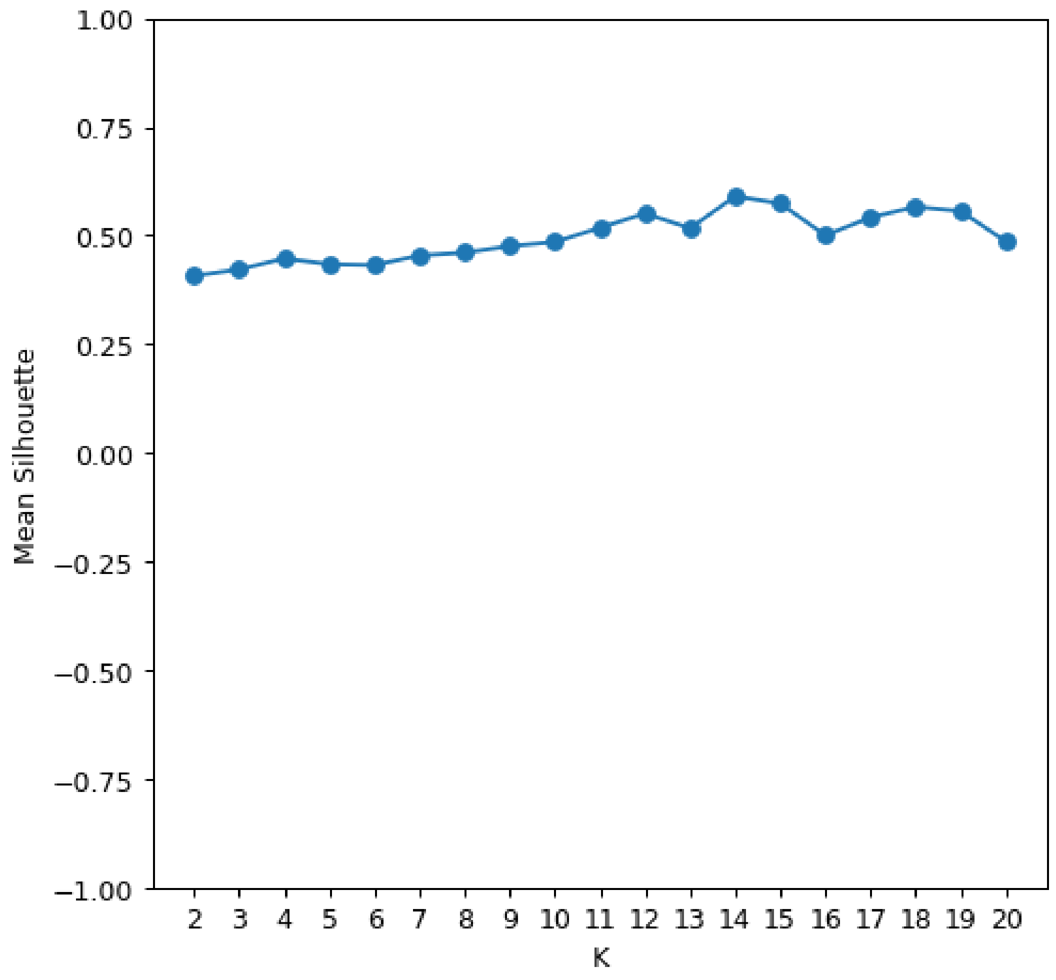
Let's analyze how our average silhouette varies with different K values for the Gaussian clusters.

[19]:
```
np.random.seed(6)
scores = []
```

```python
K_values = list(range(2, 21))
for k in K_values:
    print(f"Computing K:{k}")
    kmeans_model = KMeans(k)
    labels = kmeans_model.fit_predict(X_gauss)
    scores.append(silhouette_score(X_gauss, labels))

fig, ax = plt.subplots(figsize=(6,6), dpi=90)
ax.plot(scores, marker="o")
ax.set_ylabel("Mean Silhouette")
ax.set_ylim(-1,1)

ax.set_xlabel("K")
ax.set_xticks(list(range(len(K_values))))
_ = ax.set_xticklabels(K_values)
```

```
Computing K:2
Early stopping at iteration 28!
Computing K:3
Early stopping at iteration 12!
Computing K:4
Early stopping at iteration 10!
Computing K:5
Early stopping at iteration 18!
Computing K:6
Early stopping at iteration 18!
Computing K:7
Early stopping at iteration 25!
Computing K:8
Early stopping at iteration 15!
Computing K:9
Early stopping at iteration 27!
Computing K:10
Early stopping at iteration 16!
Computing K:11
Early stopping at iteration 9!
Computing K:12
Early stopping at iteration 41!
Computing K:13
Early stopping at iteration 27!
Computing K:14
Early stopping at iteration 42!
Computing K:15
Early stopping at iteration 11!
Computing K:16
Early stopping at iteration 9!
Computing K:17
```

```
Early stopping at iteration 30!
Computing K:18
Early stopping at iteration 18!
Computing K:19
Early stopping at iteration 18!
Computing K:20
Early stopping at iteration 25!
```



The figure shows a clear trend. The average silhouette value increases almost constantly when K rails from 2 to 14. Then, a plateau is reached with no further improvements after K=15. This is what we expected from our previous analysis.

Note that here the highest value is achieved with K=14. This is possible, although the right number of cluster is 15, since we are working with euclidean distances and clusters are not completely separated from each other.