Weighted K-Means Clustering

In this exercise we will simulate finding good locations for production plants of a company in order to minimize its logistical costs. In particular, we would like to place production plants near customers so as to reduce shipping costs and delivery time.

We assume that the probability of someone being a customer is independent of its geographical location and that the overall cost of delivering products to customers is proportional to the squared Euclidean distance to the closest production plant. Under these assumptions, the K-Means algorithm is an appropriate method to find a good set of locations. Indeed, K-Means finds a spatial clustering of potential customers and the centroid of each cluster can be chosen to be the location of the plant.

Because there are potentially millions of customers, and that it is not scalable to model each customer as a data point in the K-Means procedure, we consider instead as many points as there are geographical locations, and assign to each geographical location a weight w_i corresponding to the number of inhabitants at that location. The resulting problem becomes a weighted version of K-Means where we seek to minimize the objective:

$$J(c_1, \ldots, c_K) = \frac{\sum_i w_i \min_k ||x_i - c_k||^2}{\sum_i w_i},$$

where c_k is the kth centroid, and w_i is the weight of each geographical coordinate x_i. In order to minimize this cost function, we iteratively perform the following EM computations:

- **Expectation step:** Compute the set of points associated to each centroid:

$$\forall\, 1 \leq k \leq K: \quad \mathcal{C}(k) \leftarrow \left\{ i \;:\; k = \arg\min_k ||x_i - c_k||^2 \right\}$$

- **Minimization step:** Recompute the centroid as a the (weighted) mean of the associated data points:

$$\forall\, 1 \leq k \leq K: \quad c_k \leftarrow \frac{\sum_{i \in \mathcal{C}(k)} w_i \cdot x_i}{\sum_{i \in \mathcal{C}(k)} w_i}$$

until the objective $J(c_1, \ldots, c_K)$ has converged.

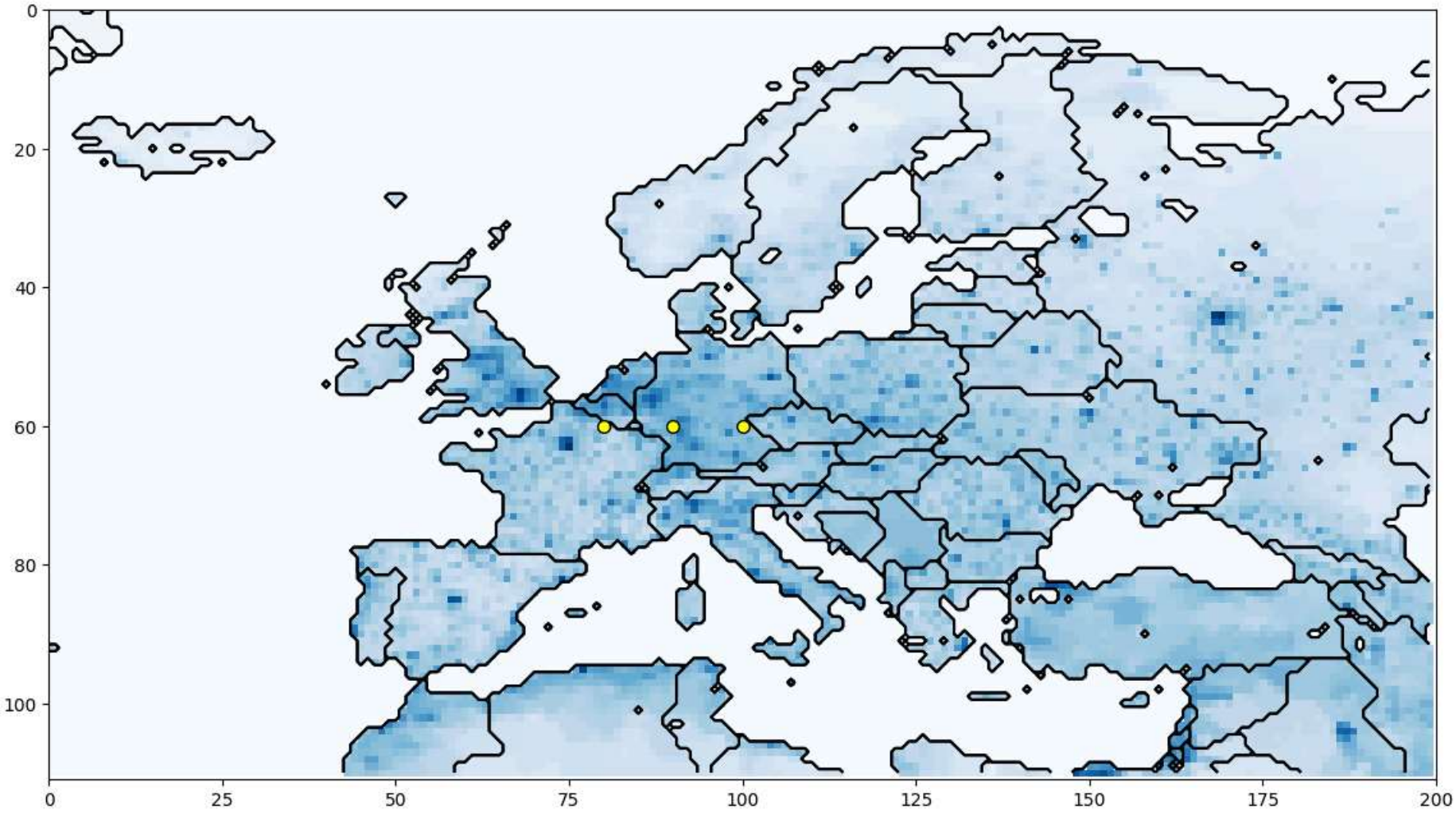
Getting started

In this exercise we will use data from http://sedac.ciesin.columbia.edu/, that we store in the files `data.mat` as part of the zip archive. The data contains for each geographical coordinates (latitude and longitude), the number of inhabitants and the corresponding country. Several variables and methods are provided in the file `utils.py`:

- **`utils.population`** A 2D array with the number of inhabitants at each latitude/longitude.
- **`utils.plot(latitudes,longitudes)`** Plot a list of centroids given as geographical coordinates in overlay to the population density map.

The code below plots three factories (white squares) with geographical coordinates (60,80), (60,90),(60,100) given as input.

In [25]:
```
import utils, numpy
%matplotlib inline
utils.plot([60,60,60],[80,90,100])
```



Also, to get a dataset of geographical coordinates associated to the image given as an array, we can use:

In [26]:
```
x, y = numpy.indices(utils.population.shape)
locations = numpy.array([x.flatten(), y.flatten()]).T
```

Initializing Weighted K-Means (25 P)

Because K-means has a non-convex objective, choosing a good initial set of centroids is important. Centroids are drawn from from the following discrete probability distribution:

$$P(x, y) = \frac{1}{Z} \cdot \text{population}(x, y)$$

where Z is a normalization constant. Furthermore, to avoid identical centroids, we add a small Gaussian noise to the location of centroids, with standard deviation 0.01.

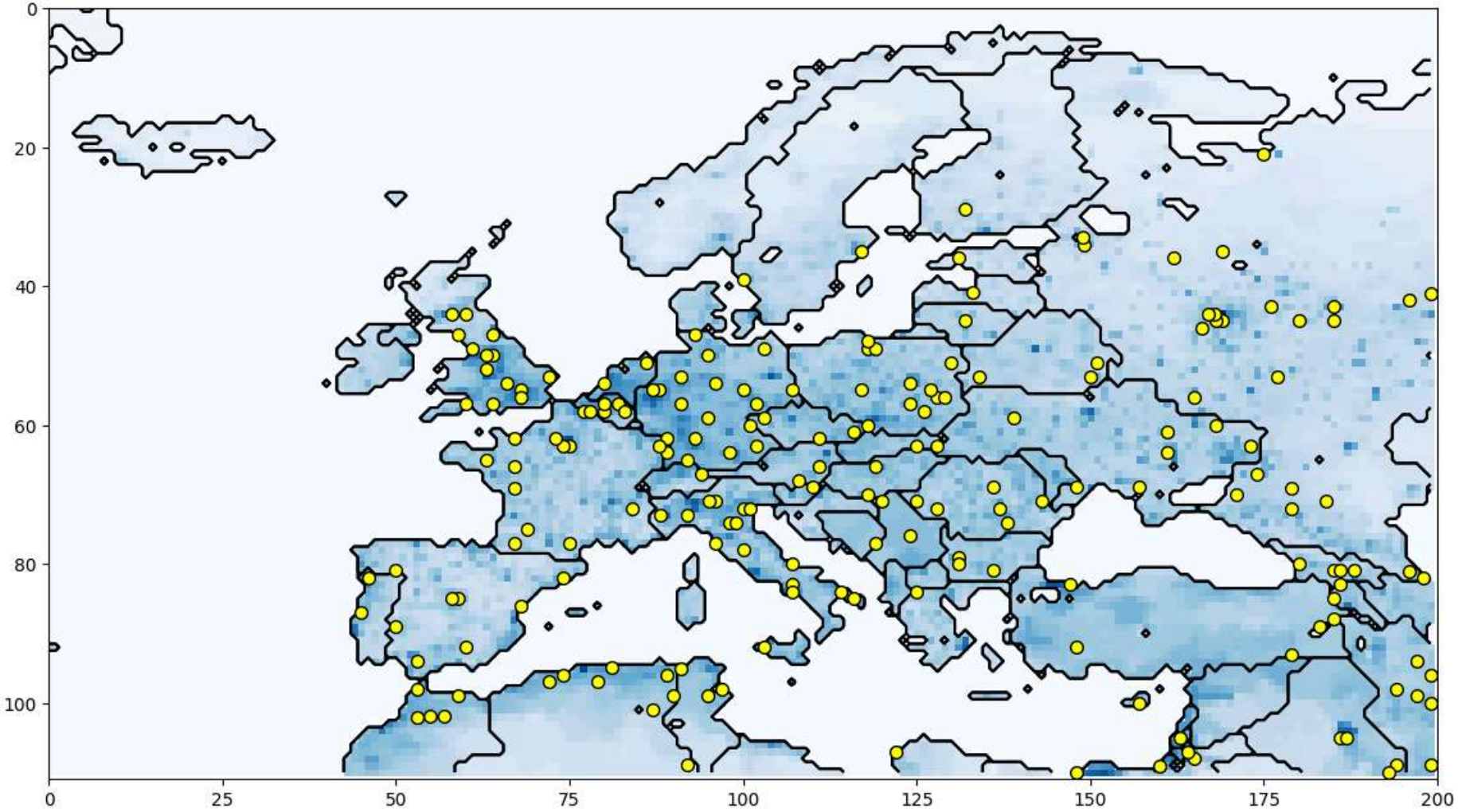
Task:

- **Implement the initialization procedure above.**

In [27]:
```
def initialize(K, population):
    # YOUR CODE HERE
    height, width = population.shape

    flat_population = population.flatten()
    Z = numpy.sum(flat_population)

    probs = flat_population/Z

    x, y = numpy.meshgrid(numpy.arange(width), numpy.arange(height))
    all_positions = numpy.column_stack((y.ravel(), x.ravel()))

    samples = numpy.random.choice(
        len(all_positions), size=K, replace=False, p=probs
    )
    sampled_centroids = all_positions[samples]

    # add noise
    noise = numpy.random.normal(loc=0.0, scale=0.01, size=sampled_centroids.shape)
    centroids = sampled_centroids + noise
    return centroids
    # --------------
```

The following code runs the initialization procedure for K=200 clusters and visualizes the centroids obtained with the initialization procedure using `utils.plot`.

In [28]:
```
centroids_init = initialize(200, utils.population)
utils.plot(centroids_init[:,0], centroids_init[:,1])
```



Implementing Weighted K-Means (75 P)

Task:

- **Implement the weighted K-Means algorithm. Your algorithm should run for `nbit` iterations and print the value of the objective after training. If `verbose`, it should also print the value of the objective at each iteration.**

In [51]:
```
def wkmeans(centroids, points, weights, verbose, nbit):
    # YOUR CODE HERE
    N, D = points.shape
    N_c, D = centroids.shape

    for iteration in range(nbit):
        # E-Step
        distances = numpy.linalg.norm(points[:, None, :] - centroids[None, :, :], axis=2)  # (N, K)
        assignments = numpy.argmin(distances, axis=1)

        # M-Step
        new_centroids = numpy.zeros_like(centroids)
        for j in range(N_c):
            assigned_points = points[assignments == j]
            assigned_weights = weights[assignments == j]
            if len(assigned_points) > 0:
                new_centroids[j] = numpy.sum(assigned_points * assigned_weights[:, None], axis=0) / numpy.sum(assigned_weights)
            else:
                new_centroids[j] = points[numpy.random.choice(N)]

        centroids = new_centroids

        # objective function value
        objective = numpy.sum([
            weights[i] * numpy.linalg.norm(points[i] - centroids[assignments[i]])**2
            for i in range(N)
        ]) / numpy.sum(weights)

        if verbose:
            print(f"Iteration {iteration + 1}: Objective = {objective}")

    print(f"Final Objective = {objective}")
    return centroids
    # --------------
```

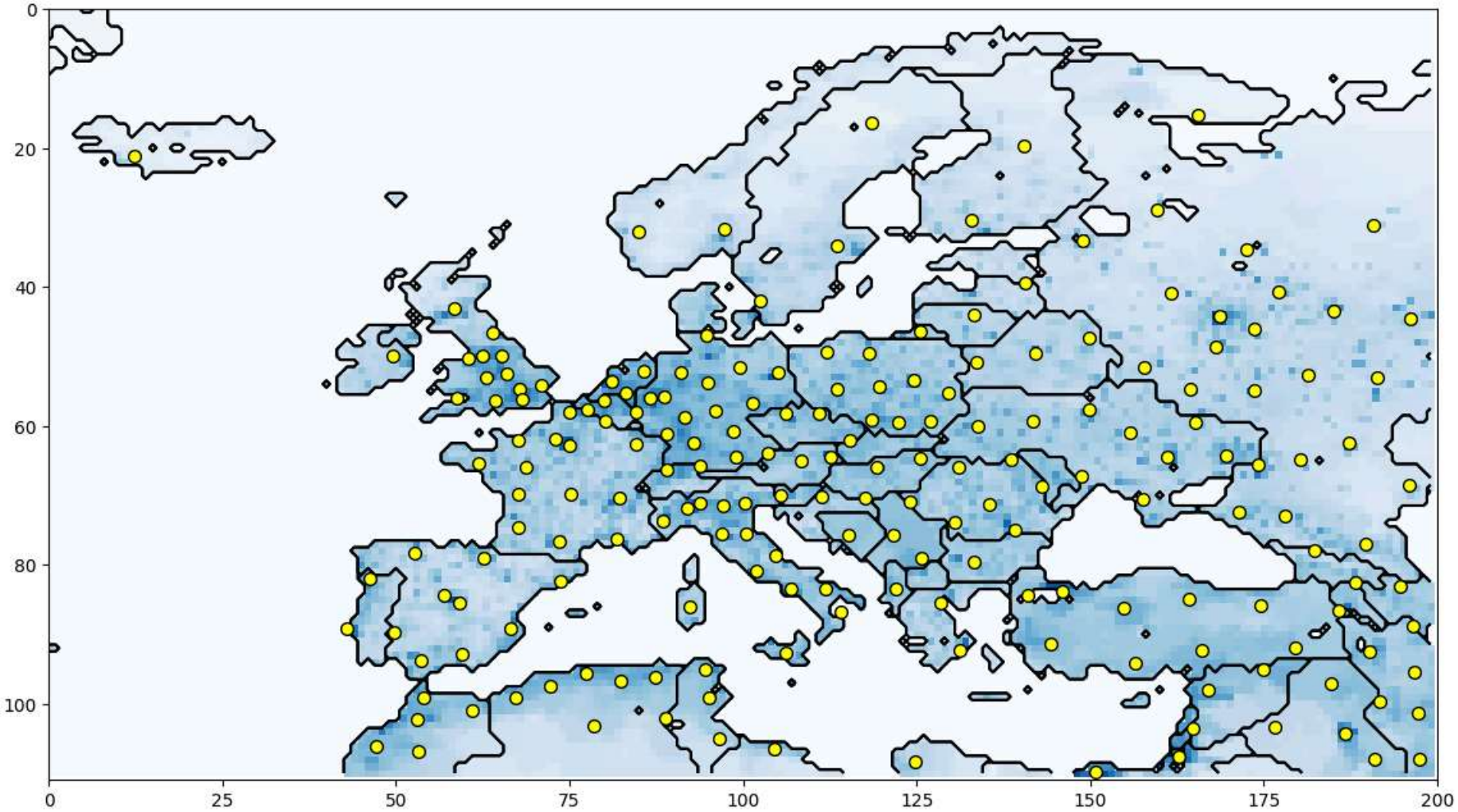
The following code runs the weighted k-means on this data, and displays the final centroids.

In [52]:
```
weights   = utils.population.flatten()*1.0
```

```
centroids = wkmeans(centroids_init, locations, weights, True, 50)

utils.plot(centroids[:,0], centroids[:,1])
```

```
Iteration 1: Objective = 13.305731618019449
Iteration 2: Objective = 9.929446109105978
Iteration 3: Objective = 8.68090640882125
Iteration 4: Objective = 7.848987856462861
Iteration 5: Objective = 7.532667989212895
Iteration 6: Objective = 7.351025598392323
Iteration 7: Objective = 7.19120882109981
Iteration 8: Objective = 7.077040080866086
Iteration 9: Objective = 7.006531184275362
Iteration 10: Objective = 6.9595018166011045
Iteration 11: Objective = 6.925764224915943
Iteration 12: Objective = 6.893378245286334
Iteration 13: Objective = 6.863788935868186
Iteration 14: Objective = 6.843386381384613
Iteration 15: Objective = 6.823826301912921
Iteration 16: Objective = 6.7977434819398965
Iteration 17: Objective = 6.766362190948664
Iteration 18: Objective = 6.743046157166166
Iteration 19: Objective = 6.725935975820578
Iteration 20: Objective = 6.7131968553135115
Iteration 21: Objective = 6.693862903575792
Iteration 22: Objective = 6.6711324771827725
Iteration 23: Objective = 6.653899901676936
Iteration 24: Objective = 6.6420867976284965
Iteration 25: Objective = 6.629245656406176
Iteration 26: Objective = 6.61949683886332
Iteration 27: Objective = 6.604520522509784
Iteration 28: Objective = 6.598207378047099
Iteration 29: Objective = 6.59493113508941
Iteration 30: Objective = 6.588522876168022
Iteration 31: Objective = 6.585633098125308
Iteration 32: Objective = 6.583994363695731
Iteration 33: Objective = 6.583083461152127
Iteration 34: Objective = 6.581369059502809
Iteration 35: Objective = 6.578423573629938
Iteration 36: Objective = 6.577780096442776
Iteration 37: Objective = 6.57736533932845
Iteration 38: Objective = 6.577128094528565
Iteration 39: Objective = 6.576828091046126
Iteration 40: Objective = 6.575900520218642
Iteration 41: Objective = 6.5747312148874295
Iteration 42: Objective = 6.573708687972192
Iteration 43: Objective = 6.572437592870369
Iteration 44: Objective = 6.570303480562838
```

```
Iteration 45: Objective = 6.5682225305515445
Iteration 46: Objective = 6.567451536063561
Iteration 47: Objective = 6.5669916707328975
Iteration 48: Objective = 6.5667433754036155
Iteration 49: Objective = 6.566638762082034
Iteration 50: Objective = 6.56658031331832
Final Objective = 6.56658031331832
```



Observe that the k-means algorithm is non-convex, and arrives in local optima of different quality depending on the initialization:

In [53]:
```
for i in range(5):
    wkmeans(initialize(200, utils.population), locations, weights, False, 50)
```

```
Final Objective = 6.633270961274992
Final Objective = 6.693957470583957
Final Objective = 7.7355085112382485
Final Objective = 6.96906757943099
Final Objective = 7.634015601805338
```

In [ ]: