Consider a small "city" of people.

- Each point represents a person
- Friendships are formed entirely based on how close they live to each other

Could you put these people into communities?

How would you tell a program to do what you did visually?

Remember how the computer "sees" these points
How do we formalize what we did visually?

- Let's assume for now that we know there are exactly two communities
- How can we assign each person to a community?
- Naive idea: Randomly assign points to each community

```python
from sklearn.utils import check_random_state
def get_random_assignment(random_state=None):
    rng = check_random_state(random_state)
    y = rng.randint(2, size=X.shape[0])
    return y

y_rand = get_random_assignment(random_state=0)
plt.scatter(X[:, 0], X[:, 1], c=y_rand, s=50, cmap='viridis')
```

This clustering "looks" quite bad.
How can we formalize whether a particular assignment is good or bad?

- One intuition: People in a communities will be as close to each other as possible.
- Take average distance between each person in a community to every other person in the same community.
- Sum over all communities.

Implement objective in via vectorized calls

\[
C_j = \{x \in X : y = j\}
\]

\[
\sum_{j=1}^{k} \frac{1}{2|C_j|} \sum_{x \in C_j, z \in C_j} \text{dist}(x, z)^2
\]

In [5]:
```python
from sklearn.metrics import pairwise_distances
# Using vectorized and list comprehensions computation
def objective(X, y):
    y_vals = np.unique(y)
    def inner(yv):
        sel = (y==yv)  # boolean array
        Xj = X[sel, :]
        n_community = np.sum(sel)
        community_sum = np.sum(pairwise_distances(Xj, Xj)**2)
        return community_sum / (2*n_community)
    return np.sum([inner(yv) for yv in y_vals])

print(objective(X, y_rand))
```

767.2572924351311

Intuition sanity check, does visual clustering solution have a low value?
Clustering goal: Minimize objective over possible community assignments

\[
\arg \min_{c_1, c_2} \sum_{j=1}^{k} \frac{1}{2|C_j|} \sum_{x \in C_j, z \in C_j} \text{dist}(x, z)^2
\]

- Naively, we could just enumerate all possibilities
- Let's try several random combinations
How many possible assignments are there?

In terms of the number of samples $n$ and the number of communities $k$

```
In [8]: n_samples = X.shape[0]
n_communities = 2
n_assignments = n_communities ** (n_samples - 1)
print('For %d samples and %d communities, there are %d possible assignments' % (n_samples, n_communities, n_assignments))
print('Or in exponential notation: %g possible assignments' % n_assignments)
```

For 200 samples and 2 communities, there are 803469022129495137770981046170581301261101496891396417650688 possible assignments
Or in exponential notation: 8.03469e+59 possible assignments
Some perspective: Fastest super computer is 200 petaflops = $2 \times 10^{17}$ operations per second

```python
In [9]:
    ops = 2 * (10 ** 17)
    print(ops)
    compute_time = n_assignments / ops
    compute_time_years = compute_time / 60 / 60 / 24 / 365
    print('Years of compute time: %d' % compute_time_years)

200000000000000000
Years of compute time: 127389177785625178899305200808361984
```

Clearly, not a good way to optimize

Let's consider a equivalent optimization

Can you figure out what these two equations mean?

$$
\mu_j \equiv \frac{1}{|C_j|} \sum_{x \in C_j} x_i
$$

$$
\arg\min_{c_1, c_2, \ldots, c_k} \sum_{j=1}^k \sum_{x \in C_j} \text{dist}(x, \mu_j)^2
$$

Consider an equivalent optimization via community representatives

- Intuition: Instead of measuring from each person to every other person in the same community, measure between a person and an ideal "representative" of each community, who is at the center of everyone.
- Representative can move freely.
- If the community assignments $C_j$ are fixed, then the position of the "representative", denoted by $\mu_j$ is defines as the mean/average point:

$$
\mu_j \equiv \frac{1}{|C_j|} \sum_{x \in C_j} x_i
$$

- Given this definition of the representative, this leads to the following equivalent minimization:
\[
\text{arg} \min_{c_1, c_2, \ldots, c_k} \sum_{j=1}^{k} \sum_{x \in c_j} \text{dist}(x, \mu_j)^2
\]

\[
\text{arg} \min_{c_1, c_2, \ldots, c_k} \sum_{j=1}^{k} \sum_{x \in c_j} \text{dist} \left( x, \frac{1}{|c_j|} \sum_{x \in c_j} x \right)^2
\]

(Derivation of equivalence can be seen at https://www.math.ucdavis.edu/~strohmer/courses/180BigData/180lecture_kmeans.pdf)

**Implement the objective of the equivalent optimization**

\[
\text{arg} \min_{c_1, c_2, \ldots, c_k} \sum_{j=1}^{k} \sum_{x \in c_j} \text{dist}(x, \mu_j)^2
\]

In [11]:

```python
def objective2(X, y):
    k = len(np.unique(y))
    out = 0
    for j in range(k):
        sel = (y==j)  # boolean array
        Xj = X[sel, :]
        mu_j = np.mean(Xj, axis=0)
        dist_to_mu = np.sqrt(np.sum((Xj - mu_j)**2, axis=0))
        out += np.sum(dist_to_mu)**2
    return out

print('Quick sanity check that objective corresponds to visual understanding')
print('Objective random', objective2(X, y_rand))
print('Objective visual', objective2(X, y_true))
```

Quick sanity check that objective corresponds to visual understanding
Objective random 767.679899871254
Objective visual 94.67363954089788

**Let's suppose the representative can move around and the communities haven't settled yet**

\[
\text{arg} \min_{c_1, \ldots, c_k, \mu_1, \ldots, \mu_k} \sum_{j=1}^{k} \sum_{x \in c_j} \text{dist}(x, \mu_j)^2
\]

- Two intuitive ideas in this "unsettled" state
  1. People will join the community of their closest representative \( \mu_j \).
     \[ y_i = \text{arg} \min_{j=1,2,\ldots,k} \text{dist}(x_i, \mu_j) \]
2. The representative will move to the center of its current community.

\[ \mu_j = \frac{1}{|C_j|} \sum_{x \in C_j} x \]

In [12]:
```python
def objective3(X, y, mu_array):
    k = len(np.unique(y))
    out = 0
    for j in range(k):
        sel = (y == j)  # boolean array
        Xj = X[sel, :]
        mu_j = mu_array[j, :]
        dist_to_mu = np.sqrt(np.sum((Xj - mu_j)**2, axis=0))
        out += np.sum(dist_to_mu**2)
    return out
```

Two intuitive ideas in this "unsettled" state

1. People will join the community of their closest representative \( \mu_j \).

\[ y_i = \arg \min_{j=1,2,\ldots,k} \text{dist}(x_i, \mu_j) \]

2. The representative will move to the center of its current community.

Let's assume the representatives don't know anything about the community so they just randomly choose to start in one house

(1) Assign people to their communities based on the representatives

In [13]:
```python
mu_array = np.array([[0, 1], [1, 0]])
print(objective3(X, y_rand, mu_array))
```

```
def best_assignment(X, mu_array):
    y_best = np.argmin(pairwise_distances(X, mu_array), axis=1)
    return y_best
```

```python
y_new = best_assignment(X, mu_array)
print(objective3(X, y_new, mu_array))
```

```
1962.992539917816
1482.1076321431726
```
Make simple function for plotting (use ax as argument)

```
In [14]: def plot_clustering(X, y, mu_array, ax=None):
    if ax is None:
        ax = plt.gca()
    ax.plot(mu_array[:, 0], mu_array[:, 1], 'ro', markersize=10)
    ax.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
    ax.set_title('Objective = %.2f' % objective3(X, y, mu_array))

fig, axes = plt.subplots(1, 2, figsize=(12, 4))
for ycur, ax in zip([y_rand, y_new], axes):
    plot_clustering(X, ycur, mu_array, ax=ax)
```

(2) Now let's move the representative to the center of its community
Def recenter(X, y):
return np.array([np.mean(X[y==yv, :], axis=0)
    for yv in np.unique(y)]
mu_array_new = recenter(X, y_new)

fig, axes = plt.subplots(1, 2, figsize=(12, 4))
for m, ax in zip([mu_array, mu_array_new], axes):
    plot_clustering(X, y_new, m, ax=ax)

What do you think you should do next?

# Program kmeans
def kmeans_alg(X, maxiter=100, random_state=None):
rng = check_random_state(random_state)

# Initialize with random points in X
rand_idx = rng.permutation(X.shape[0])
mu_array = X[rand_idx[:2], :]
y = get_random_assignment(random_state=rng)

for i in range(maxiter):
    # Get new best assignment
    y_old = y   # Save old assignment matrix
    y = best_assignment(X, mu_array)

    # Recenter / compute cluster mean
    mu_array = recenter(X, y)

    # Check convergence
    if y_old is not None and np.all(y == y_old):
        print('Converged after %d iteration' % i)
        break
return y, mu_array
Let's inspect the underlying operation by splitting the iteration
The code snippet shows the implementation of the k-means algorithm in scikit-learn. The function `kmeans_alg` is defined to perform k-means clustering. The function takes an input array `X`, a maximum number of iterations `maxiter`, and a random state for reproducibility. It initializes cluster centers with random points from the input data, iterates to update the cluster assignments and centroids until convergence, and returns the final assignments and centroids.

The code also demonstrates the usage of this function with a sample dataset. The resulting assignments and centroids are visualized using matplotlib, showing the convergence of the algorithm after 3 iterations.

The output text includes a titled introduction to scikit-learn's `sklearn.cluster.KMeans` class.
• Documentation: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html (some nice examples at the bottom of the documentation)
• See Python handbook for nice examples of kmeans https://jakevdp.github.io/PythonDataScienceHandbook/05.11-k-means.html

```
In [20]: from sklearn.datasets.samples_generator import make_blobs
   X, y_true = make_blobs(n_samples=200, centers=2,
                        cluster_std=0.50, random_state=0)

   from sklearn.cluster import KMeans
   kmeans = KMeans(n_clusters=2, random_state=0)  # 0 and 2 give opposite clust
   kmeans.fit(X)
   y_kmeans = kmeans.labels_
   mu_array = kmeans.cluster_centers_
   plot_clustering(X, y_kmeans, mu_array)

Objective = 94.6736
```

This looks great! But isn't this an NP-Hard problem?

First caveat: Does not always converge to the optimal/best solution.
Second caveat: Choosing the number of clusters is not obvious
Third caveat: Scaling of variables and clusters matters
Fourth caveat: Only linear boundaries between clusters

```
In [23]: from sklearn.datasets import make_moons
X3, y_true = make_blobs(n_samples=300, centers=2,
                       cluster_std=0.60, random_state=0)
X3[:, 0] = X3[:, 0] * 10

kmeans = KMeans(n_clusters=2, random_state=0).fit(X3)
plot_clustering(X3, kmeans.labels_, kmeans.cluster_centers_)
plt.axis('equal')

Out[23]: (-7.813474526935023, 38.988255947105756, -1.2918715239530854, 5.9043323952750475)
```
In [24]:
from sklearn.datasets import make_moons
X4, y_true4 = make_moons(200, noise=.05, random_state=0)
kmeans = KMeans(n_clusters=2, random_state=0).fit(X4)
plot_clustering(X4, kmeans.labels_, kmeans.cluster_centers_)

Objective = 79.6623