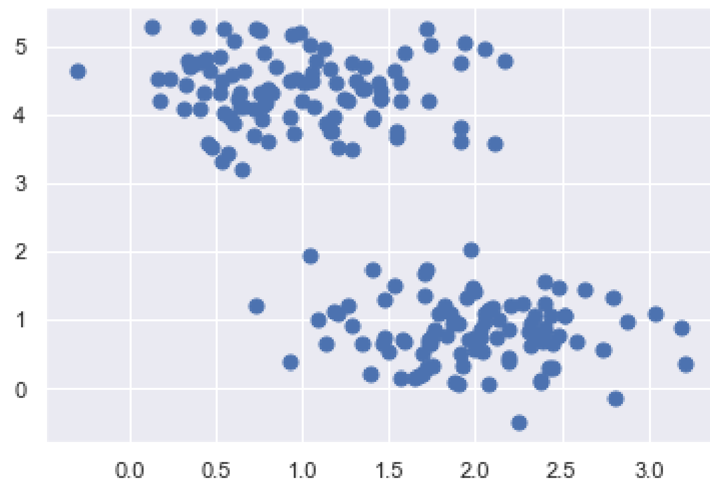
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

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?

In [2]:
```
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)
plt.scatter(X[:, 0], X[:, 1], s=50);
```



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

Remember how the computer "sees" these points

In [3]:
```python
# Print first 15 points
print(X[:15, :])
```

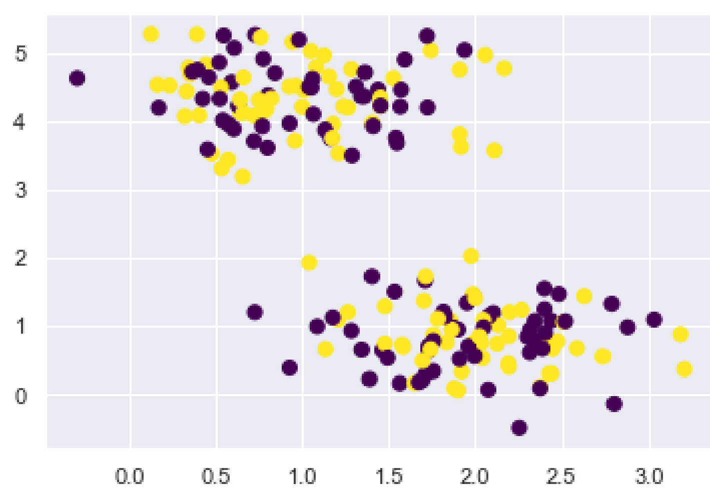
```
[[2.43859911 1.07581007]
 [1.85554301 1.0826916 ]
 [2.58952222 0.67097076]
 [1.73654901 0.69902775]
 [1.74265969 5.03846671]
 [0.64003985 4.12401075]
 [1.04829186 5.03092408]
 [0.5323772  3.31338909]
 [1.98882723 0.74876822]
 [0.16117091 4.53517846]
 [1.7571105  0.87138001]
 [1.28486901 0.92929466]
 [1.16448284 3.75408693]
 [0.3498724  4.69253251]
 [2.10413001 1.1891405 ]]
```

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

In [4]:
```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')
```

Out[4]:
```
<matplotlib.collections.PathCollection at 0x1a21957748>
```



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

$$\mathcal{C}_j = \{x \in \mathcal{X} : y = j\}$$
$$\sum_{j=1}^{k} \frac{1}{2|\mathcal{C}_j|} \sum_{x \in \mathcal{C}_j, z \in \mathcal{C}_j} \text{dist}(x, z)^2$$

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))
```

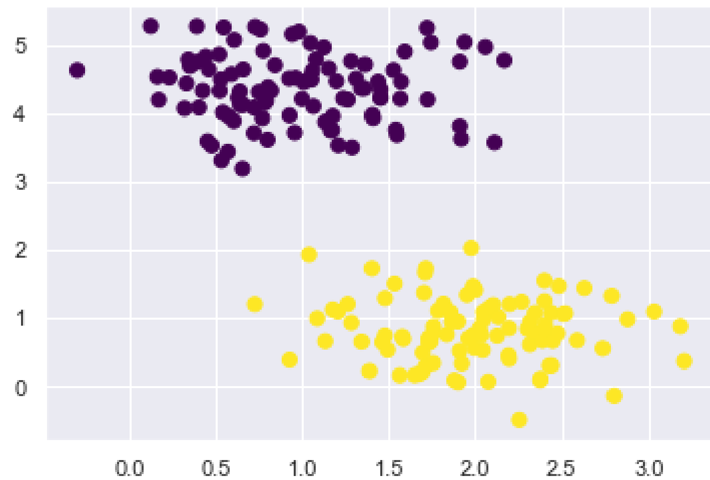
```
767.2572924351311
```

Intuition sanity check, does visual clustering solution have a low value?

In [6]:
```python
print(objective(X, y_true))
plt.scatter(X[:, 0], X[:, 1], c=y_true, s=50, cmap='viridis')
```

```
94.67363954089785
```

Out[6]: `<matplotlib.collections.PathCollection at 0x1a2309c9e8>`



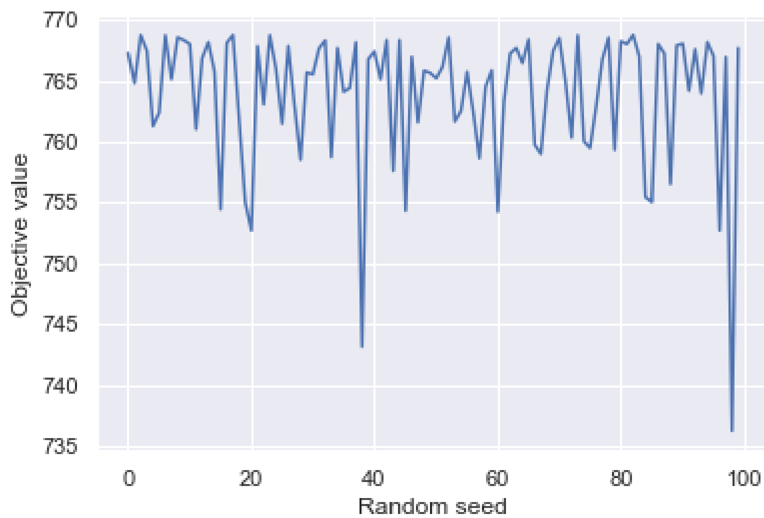
Clustering goal: Minimize objective over possible community assignments

$$\arg\min_{C_1, C_2} \sum_{j=1}^{k} \frac{1}{2|C_j|} \sum_{x \in C_j, z \in C_j} \mathrm{dist}(x, z)^2$$

- Naively, we could just enumerate all possibilities
- Let's try several random combinations

In [7]:
```python
rand_obj = np.nan * np.ones(100)
for seed in range(rand_obj.shape[0]):
    y_rand = get_random_assignment(random_state=seed)
    rand_obj[seed] = objective(X, y_rand)
    #print('Seed = %2d, Objective = %g' % (seed, obj))
plt.plot(rand_obj)
plt.xlabel('Random seed')
plt.ylabel('Objective value')
#plt.scatter(X[:, 0], X[:, 1], c=y_rand, s=50, cmap='viridis')
```

Out[7]:
```
Text(0, 0.5, 'Objective value')
```



How many possible assignments are there?

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

In [8]:
```python
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 * 10^17 operations per second

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: 12738917778562517889930520080836 1984
```

Clearly, not a good way to optimize

Let's consider a *equivalent* optimization

Can you figure out what these two equations mean?

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

$$\arg \min_{C_1, C_2, \ldots, C_k} \sum_{j=1}^{k} \sum_{x \in C_j} \text{dist}(x, \mu_j)^2$$

In [10]:
```
# Just space holder
```

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 μ_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:

$$\arg \min_{C_1, C_2, \ldots, C_k} \sum_{j=1}^{k} \sum_{x \in C_j} \text{dist}(x, \mu_j)^2$$

$$\arg \min_{C_1, C_2, \ldots, C_k} \sum_{j=1}^{k} \sum_{x \in C_j} \text{dist}\left(x, \frac{1}{|C_j|} \sum_{x \in C_j} x_i\right)^2$$

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

Implement the objective of the equivalent optimization

$$\arg \min_{C_1, C_2, \ldots, C_k} \sum_{j=1}^{k} \sum_{x \in C_j} \text{dist}(x, \mu_j)^2$$

In [11]:

```
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 understandin
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

$$\arg \min_{C_1, \ldots, C_k, \mu_1, \ldots, \mu_k} \sum_{j=1}^{k} \sum_{x \in C_j} \text{dist}(x, \mu_j)^2$$

- Two intuitive ideas in this "unsettled" state

  1. People will join the community of their closest *representative* μ_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 it's current community.

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

In [12]:
```
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* μ_j.

$$y_i = \arg \min_{j=\{1,2,\ldots,k\}} \operatorname{dist}(x_i, \mu_j)$$

2. The representative will move to the center of it's current community.

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

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

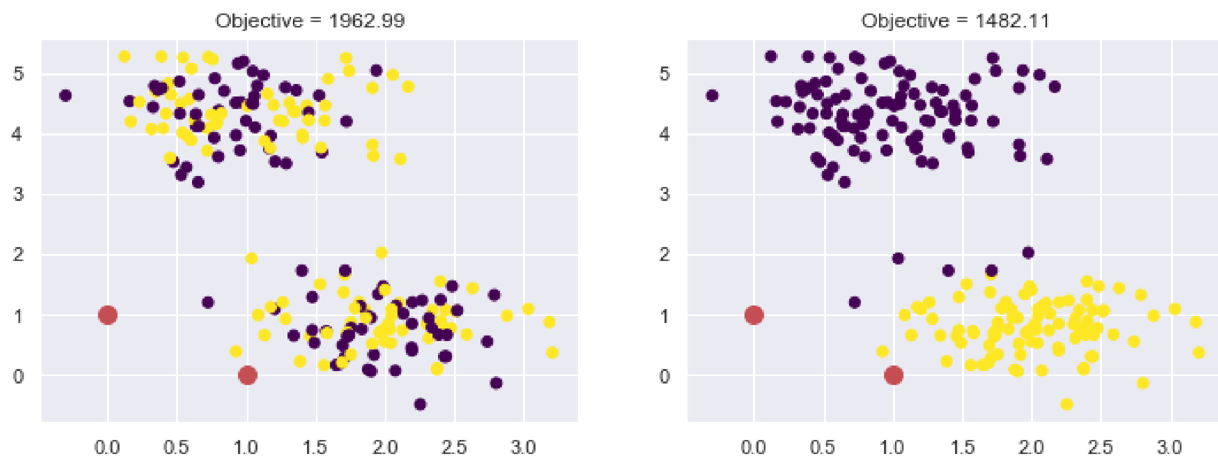
(1) Assign people to their communities based on the representatives

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

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

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

```
1962.992539917816
1482.1076321431726
```

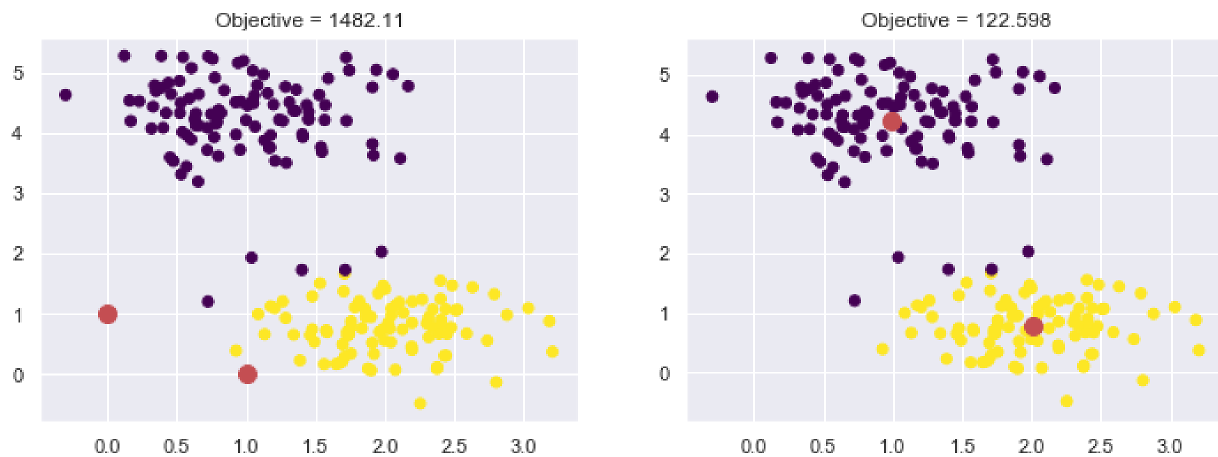
Make simple function for plotting (use ax as argument)

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 = %g' % 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

In [15]:
```python
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)
```



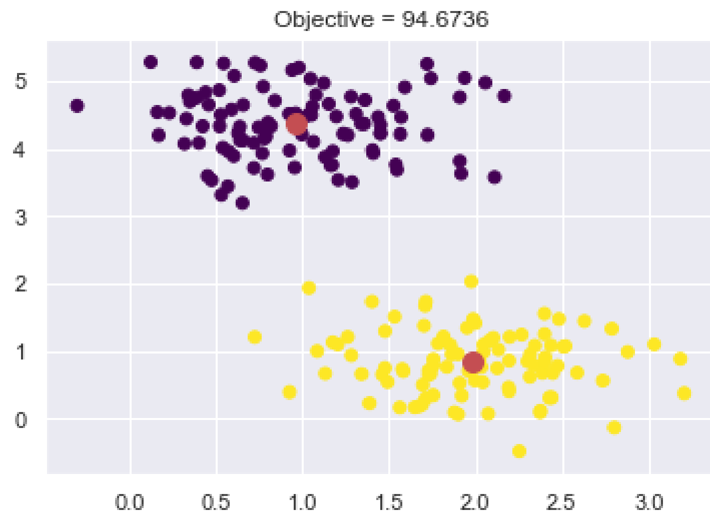
What do you think you should do next?

In [16]:
```python
# 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
```

In [17]:
```
y_kmeans, mu_kmeans = kmeans_alg(X, maxiter=100, random_state=0)

plt.plot(mu_kmeans[:, 0], mu_kmeans[:, 1], 'ro', markersize=10)
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis')
plt.title('Objective = %g' % objective3(X, y_kmeans, mu_kmeans))
```

```
Converged after 3 iteration
```

Out[17]:
```
Text(0.5, 1.0, 'Objective = 94.6736')
```



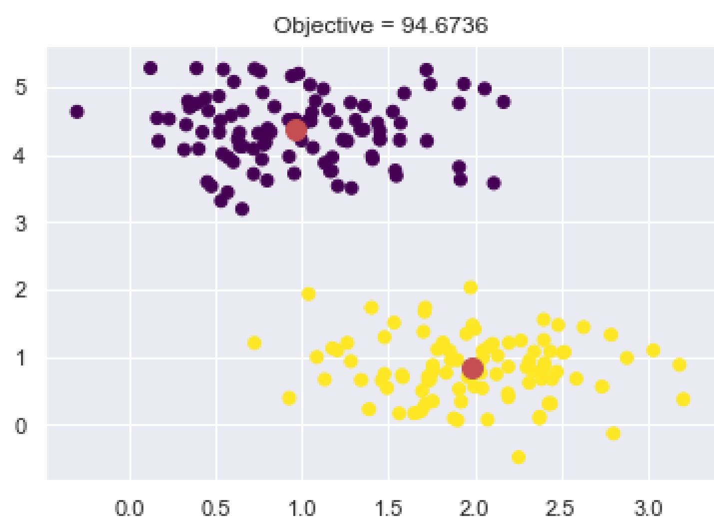
Let's inspect the underlying operation by splitting the iteration

In [18]:
```python
# 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(int(2*maxiter)): #CHANGED
        if i % 2 == 0: #CHANGED
            # Get new best assignment
            y_old = y  # Save old assignment matrix
            y = best_assignment(X, mu_array)
        else: #CHANGED
            # 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/2)) #CHANGED
                break
    return y, mu_array
```

In [19]:
```python
y_kmeans, mu_kmeans = kmeans_alg(X, maxiter=4, random_state=0)

plt.plot(mu_kmeans[:, 0], mu_kmeans[:, 1], 'ro', markersize=10)
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis')
plt.title('Objective = %g' % objective3(X, y_kmeans, mu_kmeans))
```

```
Converged after 3 iteration
```

Out[19]:
```
Text(0.5, 1.0, 'Objective = 94.6736')
```

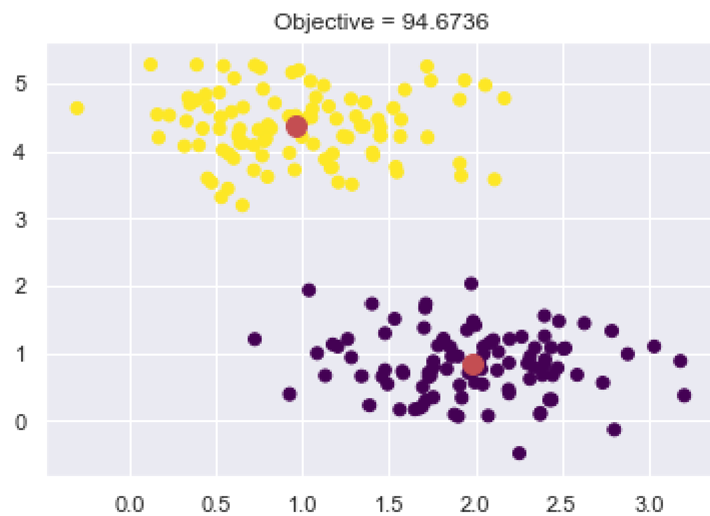


Introducing scikit-learn's `sklearn.cluster.KMeans`

- Documentation: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html (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 (https://jakevdp.github.io/PythonDataScienceHandbook/05.11-k-means.html)

In [20]:

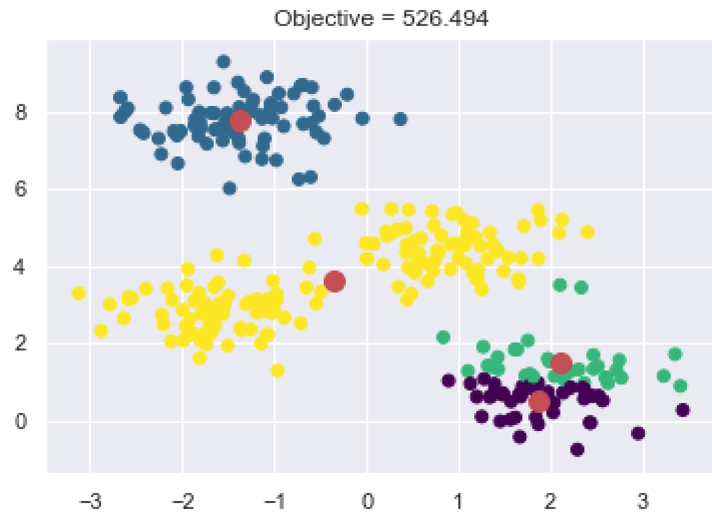
```
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=200, centers=2,
                       cluster_std=0.50, random_state=0)

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)
```



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

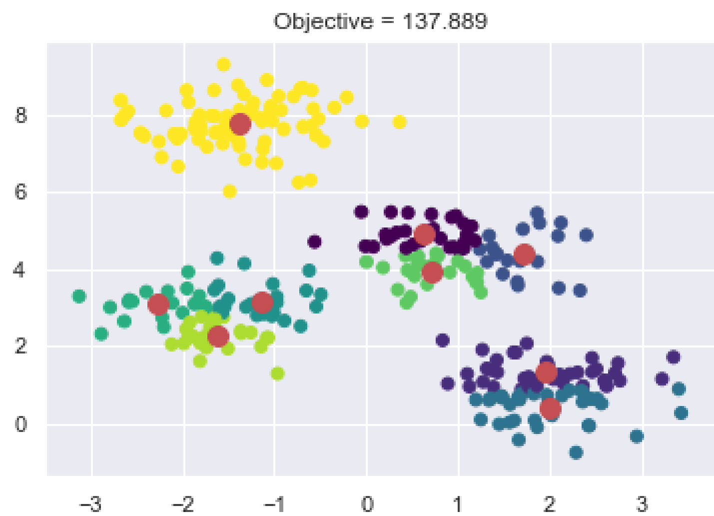
First caveat: Does not always converge to the optimal/best solution.

In [21]:
```python
# Example from Python handbook
from sklearn.datasets.samples_generator import make_blobs
X2, y_true2 = make_blobs(n_samples=300, centers=4,
                         cluster_std=0.60, random_state=0)

kmeans = KMeans(n_clusters=4, init='random', n_init=1, random_state=104) #
kmeans.fit(X2)
plot_clustering(X2, kmeans.labels_, kmeans.cluster_centers_)
```



Second caveat: Choosing the number of clusters is not obvious

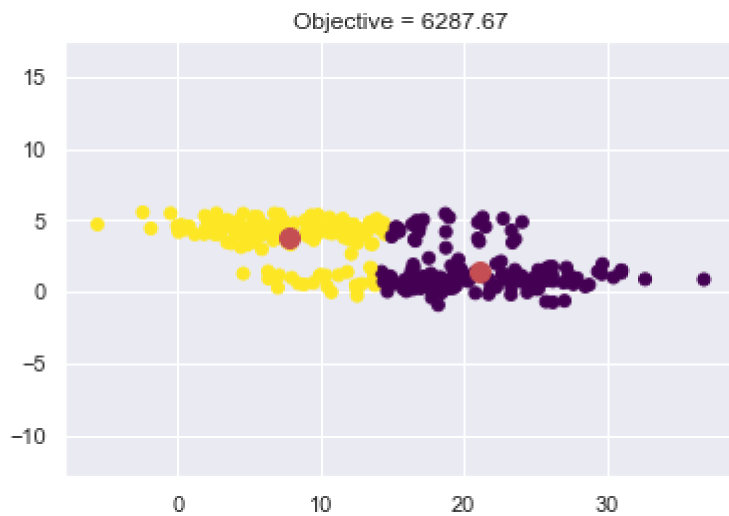
In [22]:
```python
# Example from Python handbook
from sklearn.datasets.samples_generator import make_blobs
X2, y_true2 = make_blobs(n_samples=300, centers=4,
                         cluster_std=0.60, random_state=0)

kmeans = KMeans(n_clusters=9, init='random', n_init=1, random_state=0)
kmeans.fit(X2)
plot_clustering(X2, kmeans.labels_, kmeans.cluster_centers_)
```



Third caveat: Scaling of variables and clusters matters

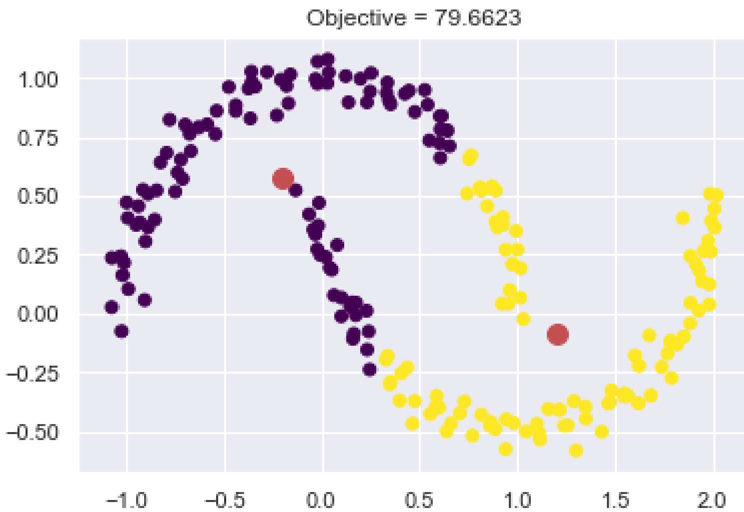
In [23]:
```python
from sklearn.datasets import make_moons
X3, y_true = make_blobs(n_samples=300, centers=2,
                        cluster_std=0.60, random_state=0)
X3[:, 0] = X3[:, 0]*10

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)
```



Fourth caveat: Only linear boundaries between clusters

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_)
```



In [ ]: