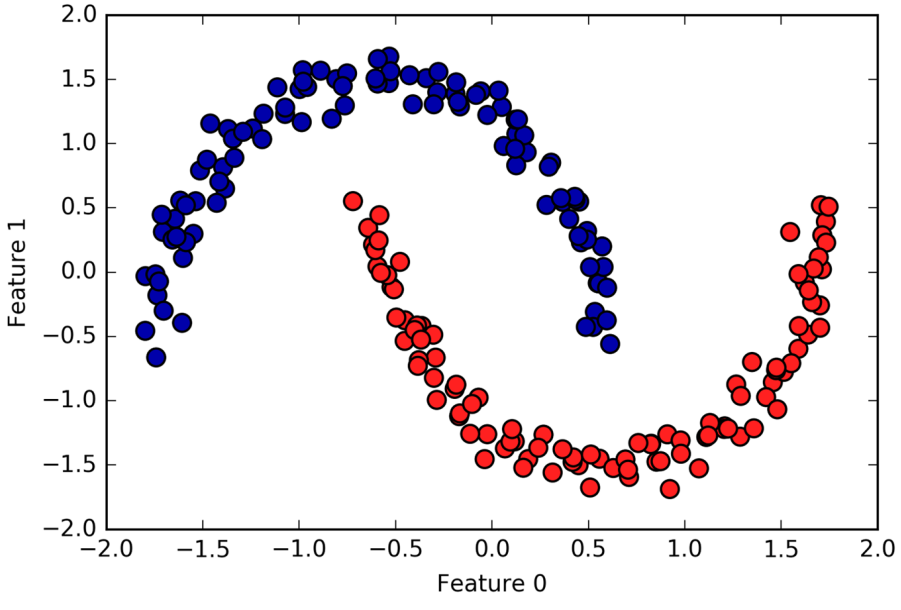


Figure 3-38. Cluster assignment found by DBSCAN using the default value of eps=0.5

Comparing and Evaluating Clustering Algorithms

One of the challenges in applying clustering algorithms is that it is very hard to assess how well an algorithm worked, and to compare outcomes between different algorithms. After talking about the algorithms behind *k*-means, agglomerative clustering, and DBSCAN, we will now compare them on some real-world datasets.

Evaluating clustering with ground truth

There are metrics that can be used to assess the outcome of a clustering algorithm relative to a ground truth clustering, the most important ones being the *adjusted rand index* (ARI) and *normalized mutual information* (NMI), which both provide a quantitative measure between 0 and 1.

Here, we compare the *k*-means, agglomerative clustering, and DBSCAN algorithms using ARI. We also include what it looks like when we randomly assign points to two clusters for comparison (see Figure 3-39):

In[68]:

```
from sklearn.metrics.cluster import adjusted_rand_score
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# rescale the data to zero mean and unit variance
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                         subplot_kw={'xticks': (), 'yticks': ()})

# make a list of algorithms to use
algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

# create a random cluster assignment for reference
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

# plot random assignment
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
                cmap=mglearn.cm3, s=60)
axes[0].set_title("Random assignment - ARI: {:.2f}".format(
        adjusted_rand_score(y, random_clusters)))

for ax, algorithm in zip(axes[1:], algorithms):
    # plot the cluster assignments and cluster centers
    clusters = algorithm.fit_predict(X_scaled)
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters,
               cmap=mglearn.cm3, s=60)
    ax.set_title("{} - ARI: {:.2f}".format(algorithm.__class__.__name__,
                                           adjusted_rand_score(y, clusters)))
```

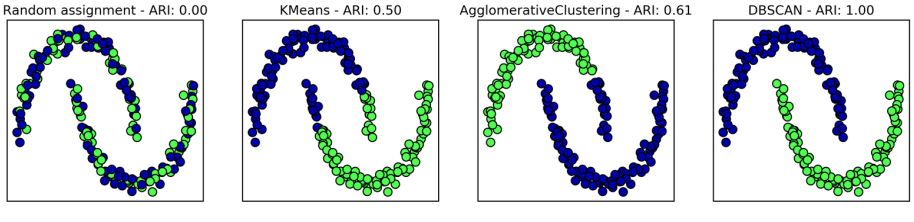


Figure 3-39. Comparing random assignment, k-means, agglomerative clustering, and DBSCAN on the two_moons dataset using the supervised ARI score

The adjusted rand index provides intuitive results, with a random cluster assignment having a score of 0 and DBSCAN (which recovers the desired clustering perfectly) having a score of 1.

A common mistake when evaluating clustering in this way is to use `accuracy_score` instead of `adjusted_rand_score`, `normalized_mutual_info_score`, or some other clustering metric. The problem in using accuracy is that it requires the assigned cluster labels to exactly match the ground truth. However, the cluster labels themselves are meaningless—the only thing that matters is which points are in the same cluster:

In[69]:

```
from sklearn.metrics import accuracy_score

# these two labelings of points correspond to the same clustering
clusters1 = [0, 0, 1, 1, 0]
clusters2 = [1, 1, 0, 0, 1]
# accuracy is zero, as none of the labels are the same
print("Accuracy: {:.2f}".format(accuracy_score(clusters1, clusters2)))
# adjusted rand score is 1, as the clustering is exactly the same
print("ARI: {:.2f}".format(adjusted_rand_score(clusters1, clusters2)))
```

Out[69]:

```
Accuracy: 0.00
ARI: 1.00
```

Evaluating clustering without ground truth

Although we have just shown one way to evaluate clustering algorithms, in practice, there is a big problem with using measures like ARI. When applying clustering algorithms, there is usually no ground truth to which to compare the results. If we knew the right clustering of the data, we could use this information to build a supervised model like a classifier. Therefore, using metrics like ARI and NMI usually only helps in developing algorithms, not in assessing success in an application.

There are scoring metrics for clustering that don't require ground truth, like the *silhouette coefficient*. However, these often don't work well in practice. The silhouette score computes the compactness of a cluster, where higher is better, with a perfect score of 1. While compact clusters are good, compactness doesn't allow for complex shapes.

Here is an example comparing the outcome of *k*-means, agglomerative clustering, and DBSCAN on the `two-moons` dataset using the silhouette score (Figure 3-40):

In[70]:

```
from sklearn.metrics.cluster import silhouette_score

X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
# rescale the data to zero mean and unit variance
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
```

```
fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                         subplot_kw={'xticks': (), 'yticks': ()})

# create a random cluster assignment for reference
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

# plot random assignment
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
    cmap=mglearn.cm3, s=60)
axes[0].set_title("Random assignment: {:.2f}".format(
    silhouette_score(X_scaled, random_clusters)))

algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

for ax, algorithm in zip(axes[1:], algorithms):
    clusters = algorithm.fit_predict(X_scaled)
    # plot the cluster assignments and cluster centers
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mglearn.cm3,
               s=60)
    ax.set_title("{} : {:.2f}".format(algorithm.__class__.__name__,
                                      silhouette_score(X_scaled, clusters)))
```

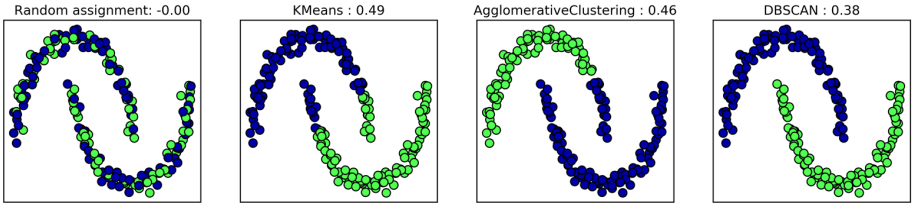


Figure 3-40. Comparing random assignment, k-means, agglomerative clustering, and DBSCAN on the two_moons dataset using the unsupervised silhouette score—the more intuitive result of DBSCAN has a lower silhouette score than the assignments found by k-means

As you can see, *k*-means gets the highest silhouette score, even though we might prefer the result produced by DBSCAN. A slightly better strategy for evaluating clusters is using *robustness-based* clustering metrics. These run an algorithm after adding some noise to the data, or using different parameter settings, and compare the outcomes. The idea is that if many algorithm parameters and many perturbations of the data return the same result, it is likely to be trustworthy. Unfortunately, this strategy is not implemented in `scikit-learn` at the time of writing.

Even if we get a very robust clustering, or a very high silhouette score, we still don't know if there is any semantic meaning in the clustering, or whether the clustering

reflects an aspect of the data that we are interested in. Let's go back to the example of face images. We hope to find groups of similar faces—say, men and women, or old people and young people, or people with beards and without. Let's say we cluster the data into two clusters, and all algorithms agree about which points should be clustered together. We still don't know if the clusters that are found correspond in any way to the concepts we are interested in. It could be that they found side views versus front views, or pictures taken at night versus pictures taken during the day, or pictures taken with iPhones versus pictures taken with Android phones. The only way to know whether the clustering corresponds to anything we are interested in is to analyze the clusters manually.

Comparing algorithms on the faces dataset

Let's apply the *k*-means, DBSCAN, and agglomerative clustering algorithms to the Labeled Faces in the Wild dataset, and see if any of them find interesting structure. We will use the eigenface representation of the data, as produced by `PCA(whiten=True)`, with 100 components:

In[71]:

```
# extract eigenfaces from lfw data and transform data
from sklearn.decomposition import PCA
pca = PCA(n_components=100, whiten=True, random_state=0)
pca.fit_transform(X_people)
X_pca = pca.transform(X_people)
```

We saw earlier that this is a more semantic representation of the face images than the raw pixels. It will also make computation faster. A good exercise would be for you to run the following experiments on the original data, without PCA, and see if you find similar clusters.

Analyzing the faces dataset with DBSCAN. We will start by applying DBSCAN, which we just discussed:

In[72]:

```
# apply DBSCAN with default parameters
dbscan = DBSCAN()
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

Out[72]:

```
Unique labels: [-1]
```

We see that all the returned labels are –1, so all of the data was labeled as "noise" by DBSCAN. There are two things we can change to help this: we can make `eps` higher, to expand the neighborhood of each point, and set `min_samples` lower, to consider smaller groups of points as clusters. Let's try changing `min_samples` first:

In[73]:

```
dbscan = DBSCAN(min_samples=3)
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

Out[73]:

```
Unique labels: [-1]
```

Even when considering groups of three points, everything is labeled as noise. So, we need to increase `eps`:

In[74]:

```
dbscan = DBSCAN(min_samples=3, eps=15)
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

Out[74]:

```
Unique labels: [-1  0]
```

Using a much larger `eps` of `15`, we get only a single cluster and noise points. We can use this result to find out what the "noise" looks like compared to the rest of the data. To understand better what's happening, let's look at how many points are noise, and how many points are inside the cluster:

In[75]:

```
# Count number of points in all clusters and noise.
# bincount doesn't allow negative numbers, so we need to add 1.
# The first number in the result corresponds to noise points.
print("Number of points per cluster: {}".format(np.bincount(labels + 1)))
```

Out[75]:

```
Number of points per cluster: [  27 2036]
```

There are very few noise points—only 27—so we can look at all of them (see Figure 3-41):

In[76]:

```
noise = X_people[labels==-1]

fig, axes = plt.subplots(3, 9, subplot_kw={'xticks': (), 'yticks': ()},
                         figsize=(12, 4))
for image, ax in zip(noise, axes.ravel()):
    ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
```

Figure 3-41. Samples from the faces dataset labeled as noise by DBSCAN

Comparing these images to the random sample of face images from Figure 3-7, we can guess why they were labeled as noise: the fifth image in the first row shows a person drinking from a glass, there are images of people wearing hats, and in the last image there's a hand in front of the person's face. The other images contain odd angles or crops that are too close or too wide.

This kind of analysis—trying to find "the odd one out"—is called *outlier detection*. If this was a real application, we might try to do a better job of cropping images, to get more homogeneous data. There is little we can do about people in photos sometimes wearing hats, drinking, or holding something in front of their faces, but it's good to know that these are issues in the data that any algorithm we might apply needs to handle.

If we want to find more interesting clusters than just one large one, we need to set `eps` smaller, somewhere between `15` and `0.5` (the default). Let's have a look at what different values of `eps` result in:

`In[77]:`

```
for eps in [1, 3, 5, 7, 9, 11, 13]:
    print("\neps={}".format(eps))
    dbscan = DBSCAN(eps=eps, min_samples=3)
    labels = dbscan.fit_predict(X_pca)
    print("Clusters present: {}".format(np.unique(labels)))
    print("Cluster sizes: {}".format(np.bincount(labels + 1)))
```

`Out[78]:`

```
eps=1
Clusters present: [-1]
Cluster sizes: [2063]

eps=3
Clusters present: [-1]
Cluster sizes: [2063]
```

```
eps=5
Clusters present: [-1]
Cluster sizes: [2063]

eps=7
Clusters present: [-1  0  1  2  3  4  5  6  7  8  9 10 11 12]
Cluster sizes: [2006  4  6  6  6  9  3  3  4  3  3  3  3  4]

eps=9
Clusters present: [-1  0  1  2]
Cluster sizes: [1269  788    3    3]

eps=11
Clusters present: [-1  0]
Cluster sizes: [ 430 1633]

eps=13
Clusters present: [-1  0]
Cluster sizes: [ 112 1951]
```

For low settings of `eps`, all points are labeled as noise. For `eps=7`, we get many noise points and many smaller clusters. For `eps=9` we still get many noise points, but we get one big cluster and some smaller clusters. Starting from `eps=11`, we get only one large cluster and noise.

What is interesting to note is that there is never more than one large cluster. At most, there is one large cluster containing most of the points, and some smaller clusters. This indicates that there are not two or three different kinds of face images in the data that are very distinct, but rather that all images are more or less equally similar to (or dissimilar from) the rest.

The results for `eps=7` look most interesting, with many small clusters. We can investigate this clustering in more detail by visualizing all of the points in each of the 13 small clusters (Figure 3-42):

`In[78]:`

```
dbscan = DBSCAN(min_samples=3, eps=7)
labels = dbscan.fit_predict(X_pca)

for cluster in range(max(labels) + 1):
    mask = labels == cluster
    n_images =  np.sum(mask)
    fig, axes = plt.subplots(1, n_images, figsize=(n_images * 1.5, 4),
                             subplot_kw={'xticks': (), 'yticks': ()})
    for image, label, ax in zip(X_people[mask], y_people[mask], axes):

        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1])
```

Figure 3-42. Clusters found by DBSCAN with eps=7

Some of the clusters correspond to people with very distinct faces (within this dataset), such as Sharon or Koizumi. Within each cluster, the orientation of the face is also

quite fixed, as well as the facial expression. Some of the clusters contain faces of multiple people, but they share a similar orientation and expression.

This concludes our analysis of the DBSCAN algorithm applied to the faces dataset. As you can see, we are doing a manual analysis here, different from the much more automatic search approach we could use for supervised learning based on R^2 score or accuracy.

Let's move on to applying *k*-means and agglomerative clustering.

Analyzing the faces dataset with k-means. We saw that it was not possible to create more than one big cluster using DBSCAN. Agglomerative clustering and *k*-means are much more likely to create clusters of even size, but we do need to set a target number of clusters. We could set the number of clusters to the known number of people in the dataset, though it is very unlikely that an unsupervised clustering algorithm will recover them. Instead, we can start with a low number of clusters, like 10, which might allow us to analyze each of the clusters:

In[79]:

```
# extract clusters with k-means
km = KMeans(n_clusters=10, random_state=0)
labels_km = km.fit_predict(X_pca)
print("Cluster sizes k-means: {}".format(np.bincount(labels_km)))
```

Out[79]:

```
Cluster sizes k-means: [269 128 170 186 386 222 237  64 253 148]
```

As you can see, *k*-means clustering partitioned the data into relatively similarly sized clusters from 64 to 386. This is quite different from the result of DBSCAN.

We can further analyze the outcome of *k*-means by visualizing the cluster centers (Figure 3-43). As we clustered in the representation produced by `PCA`, we need to rotate the cluster centers back into the original space to visualize them, using `pca.inverse_transform`:

In[80]:

```
fig, axes = plt.subplots(2, 5, subplot_kw={'xticks': (), 'yticks': ()},
                         figsize=(12, 4))
for center, ax in zip(km.cluster_centers_, axes.ravel()):
    ax.imshow(pca.inverse_transform(center).reshape(image_shape),
              vmin=0, vmax=1)
```

Figure 3-43. Cluster centers found by k-means when setting the number of clusters to 10

The cluster centers found by *k*-means are very smooth versions of faces. This is not very surprising, given that each center is an average of 64 to 386 face images. Working with a reduced PCA representation adds to the smoothness of the images (compared to the faces reconstructed using 100 PCA dimensions in Figure 3-11). The clustering seems to pick up on different orientations of the face, different expressions (the third cluster center seems to show a smiling face), and the presence of shirt collars (see the second-to-last cluster center).

For a more detailed view, in Figure 3-44 we show for each cluster center the five most typical images in the cluster (the images assigned to the cluster that are closest to the cluster center) and the five most atypical images in the cluster (the images assigned to the cluster that are furthest from the cluster center):

In[81]:

```
mglearn.plots.plot_kmeans_faces(km, pca, X_pca, X_people,
                                y_people, people.target_names)
```

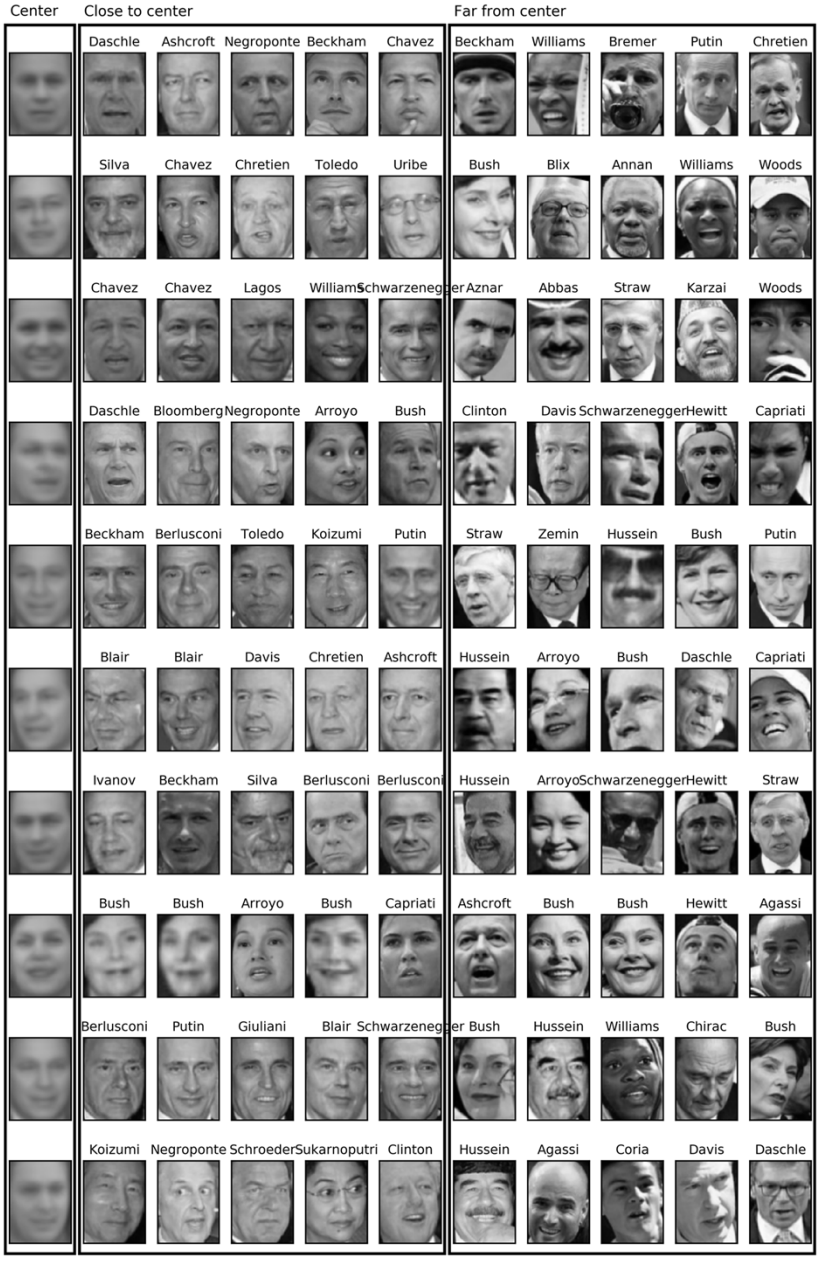


Figure 3-44. Sample images for each cluster found by k-means—the cluster centers are on the left, followed by the five closest points to each center and the five points that are assigned to the cluster but are furthest away from the center

Figure 3-44 confirms our intuition about smiling faces for the third cluster, and also the importance of orientation for the other clusters. The "atypical" points are not very similar to the cluster centers, though, and their assignment seems somewhat arbitrary. This can be attributed to the fact that *k*-means partitions all the data points and doesn't have a concept of "noise" points, as DBSCAN does. Using a larger number of clusters, the algorithm could find finer distinctions. However, adding more clusters makes manual inspection even harder.

Analyzing the faces dataset with agglomerative clustering. Now, let's look at the results of agglomerative clustering:

In[82]:

```
# extract clusters with ward agglomerative clustering
agglomerative = AgglomerativeClustering(n_clusters=10)
labels_agg = agglomerative.fit_predict(X_pca)
print("Cluster sizes agglomerative clustering: {}".format(
    np.bincount(labels_agg)))
```

Out[82]:

```
Cluster sizes agglomerative clustering: [255 623  86 102 122 199 265  26 230 155]
```

Agglomerative clustering also produces relatively equally sized clusters, with cluster sizes between 26 and 623. These are more uneven than those produced by *k*-means, but much more even than the ones produced by DBSCAN.

We can compute the ARI to measure whether the two partitions of the data given by agglomerative clustering and *k*-means are similar:

In[83]:

```
print("ARI: {:.2f}".format(adjusted_rand_score(labels_agg, labels_km)))
```

Out[83]:

```
ARI: 0.13
```

An ARI of only 0.13 means that the two clusterings `labels_agg` and `labels_km` have little in common. This is not very surprising, given the fact that points further away from the cluster centers seem to have little in common for *k*-means.

Next, we might want to plot the dendrogram (Figure 3-45). We'll limit the depth of the tree in the plot, as branching down to the individual 2,063 data points would result in an unreadably dense plot:

In[84]:

```
linkage_array = ward(X_pca)
# now we plot the dendrogram for the linkage_array
# containing the distances between clusters
plt.figure(figsize=(20, 5))
dendrogram(linkage_array, p=7, truncate_mode='level', no_labels=True)
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
```

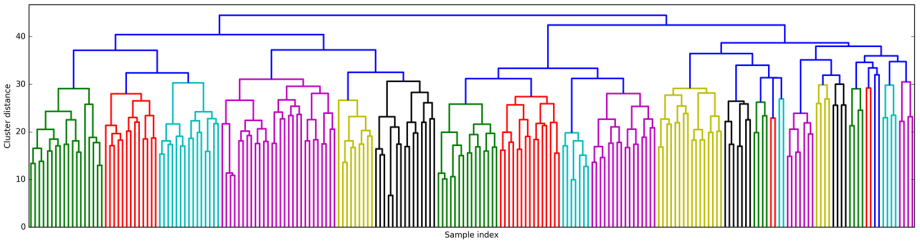


Figure 3-45. Dendrogram of agglomerative clustering on the faces dataset

Creating 10 clusters, we cut across the tree at the very top, where there are 10 vertical lines. In the dendrogram for the toy data shown in Figure 3-36, you could see by the length of the branches that two or three clusters might capture the data appropriately. For the faces data, there doesn't seem to be a very natural cutoff point. There are some branches that represent more distinct groups, but there doesn't appear to be a particular number of clusters that is a good fit. This is not surprising, given the results of DBSCAN, which tried to cluster all points together.

Let's visualize the 10 clusters, as we did for *k*-means earlier (Figure 3-46). Note that there is no notion of cluster center in agglomerative clustering (though we could compute the mean), and we simply show the first couple of points in each cluster. We show the number of points in each cluster to the left of the first image:

In[85]:

```
n_clusters = 10
for cluster in range(n_clusters):
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 10, subplot_kw={'xticks': (), 'yticks': ()},
                             figsize=(15, 8))
    axes[0].set_ylabel(np.sum(mask))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                      labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1],
                     fontdict={'fontsize': 9})
```

Figure 3-46. Random images from the clusters generated by In[82]—each row corresponds to one cluster; the number to the left lists the number of images in each cluster

While some of the clusters seem to have a semantic theme, many of them are too large to be actually homogeneous. To get more homogeneous clusters, we can run the algorithm again, this time with 40 clusters, and pick out some of the clusters that are particularly interesting (Figure 3-47):

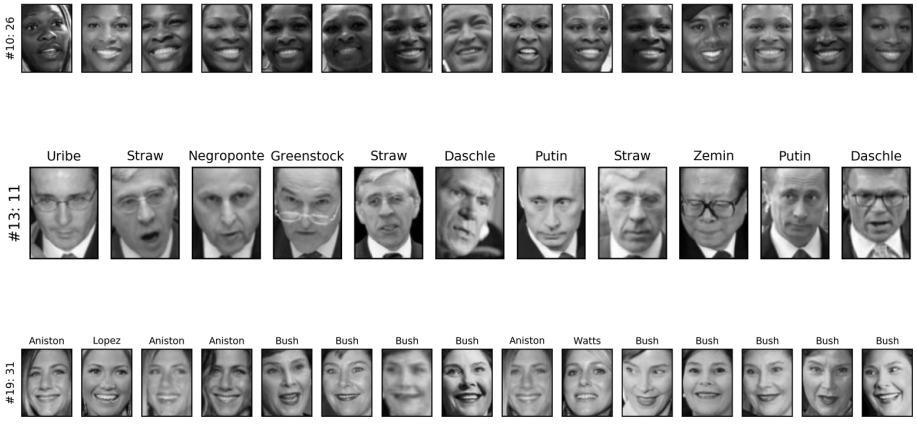
In[86]:

```
# extract clusters with ward agglomerative clustering
agglomerative = AgglomerativeClustering(n_clusters=40)
labels_agg = agglomerative.fit_predict(X_pca)
print("cluster sizes agglomerative clustering: {}".format(np.bincount(labels_agg)))

n_clusters = 40
for cluster in [10, 13, 19, 22, 36]: # hand-picked "interesting" clusters
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 15, subplot_kw={'xticks': (), 'yticks': ()},
                             figsize=(15, 8))
    cluster_size = np.sum(mask)
    axes[0].set_ylabel("#{}: {}".format(cluster, cluster_size))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                      labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1],
                     fontdict={'fontsize': 9})
    for i in range(cluster_size, 15):
        axes[i].set_visible(False)
```

Out[86]:

```
cluster sizes agglomerative clustering:
 [ 58  80  79  40 222  50  55  78 172  28  26  34  14  11  60  66 152  27
  47  31  54   5   8  56   3   5   8  18  22  82  37  89  28  24  41  40
  21  10 113  69]
```



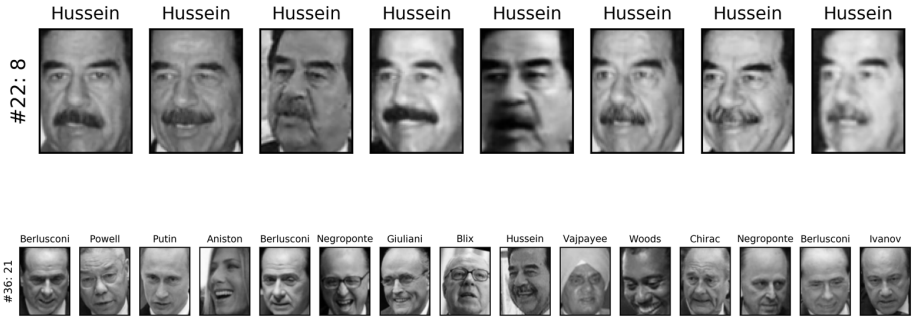


Figure 3-47. Images from selected clusters found by agglomerative clustering when setting the number of clusters to 40—the text to the left shows the index of the cluster and the total number of points in the cluster

Here, the clustering seems to have picked up on "dark skinned and smiling," "collared shirt," "smiling woman," "Hussein," and "high forehead." We could also find these highly similar clusters using the dendrogram, if we did more a detailed analysis.

Summary of Clustering Methods

This section has shown that applying and evaluating clustering is a highly qualitative procedure, and often most helpful in the exploratory phase of data analysis. We looked at three clustering algorithms: *k*-means, DBSCAN, and agglomerative clustering. All three have a way of controlling the granularity of clustering. *k*-means and agglomerative clustering allow you to specify the number of desired clusters, while DBSCAN lets you define proximity using the `eps` parameter, which indirectly influences cluster size. All three methods can be used on large, real-world datasets, are relatively easy to understand, and allow for clustering into many clusters.

Each of the algorithms has somewhat different strengths. *k*-means allows for a characterization of the clusters using the cluster means. It can also be viewed as a decomposition method, where each data point is represented by its cluster center. DBSCAN allows for the detection of "noise points" that are not assigned any cluster, and it can help automatically determine the number of clusters. In contrast to the other two methods, it allow for complex cluster shapes, as we saw in the `two_moons` example. DBSCAN sometimes produces clusters of very differing size, which can be a strength or a weakness. Agglomerative clustering can provide a whole hierarchy of possible partitions of the data, which can be easily inspected via dendrograms.