Chapter 6:
Unsupervised Learning

November 25, 2020

Meskipun kebanyakan aplikasi dari *machine learning* hari ini berdasarkan *supervised learning*, tetapi mayoritas data original tidak memiliki label. Dalam hal ini kita mempunyai input *feature* X, tetapi kita tidak memiliki label y. Seorang computer scientist yang terkenal, Yann LeCun berkata:

"***If intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake.***"

Dengan kata lain, ada potensi yang cukup besar pada *unuspervised learning* yang belum kita temukan dan dapat kita gali lebih jauh.

Jika kita ingin membuat sistem yang akan mengambil beberapa gambar pada setiap item di lini produksi manufaktur dan mendeteksi item yang rusak, kita dapat dengan mudah mengambil gambar-gambar secara otomatis, dan boleh jadi akan menghasilkan gambar beribu-ribu perhari. Dengan hasil tersebut kita dapat membuat dataset yang besar dalam waktu beberapa minggu. Tetapi, kita belum punya label untuk data-data tersebut. Jika kita ingin melatih *classifier* biner yang biasa dan dapat memprediksi bahwa sebuah item itu rusak (*defective*) atau normal, maka kita melabeli setiap gambar untuk mentraining sistem dengan label "*defective*" atau "normal". Maka hal ini membutuhkan seorang ahli untuk duduk dan secara manual melakukan inspeksi keseluruhan gambar untuk melabeli. Pekerjaan tersebut pasti akan memamkan waktu yang sangat lama, mahal dan membosankan, sehingga biasanya dilakukan untuk beberapa sampel gambar saja (tidak keseluruhan). Sehingga hasilnya, dataset yang telah dilabeli berjumlah sedikit yang mengakibatkan kinerja *classifier* yang telah ditraining dengan data terbatas tersebut tentunya akan jauh dari harapan. Lebih juah, setiap kali perusahaan ingin membuat perubahan pada produknya, keseluruhan proses harus diulangi kembali dari awal, karena jika tidak maka kinerja sistem klasifikasi yang telah dibangun sebelumnya akan sangat buruk. Oleh sebag itu, alangkah bergunanya jika sebuah algoritma dapat mengeksploitasi data tanpa label tersebut secara otomatis tanpa campur tangan manusia untuk melabeli setiap gambar. Dalam hal inilah *unsupervised learning* sangata berperan.

Pada *chapter* ini, kita akan mempelajari beberapa pekerjaan yang menyangkut *unsupervised learning* diantaranya adalah:

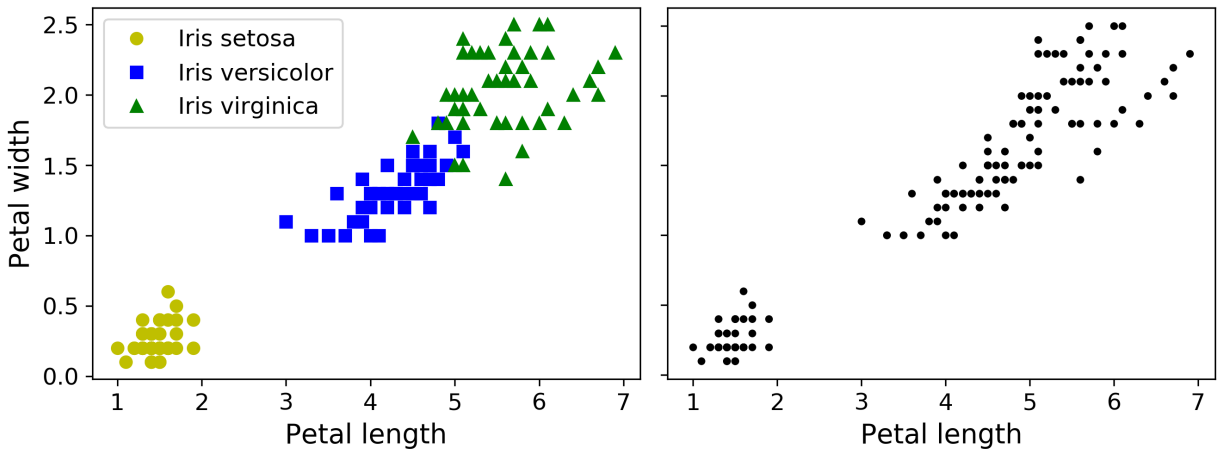
- ***Clustering***. Tujuan dari *clustering* adalah melakukan *grouping* dari *instances* yang serupa ke dalam satu *cluster*. *Clustering* adalah tool yang sangat baik untuk analisa data, segmentasi kostumer, sistem perekomendasi, *search engines*, *image segmentation*, *semi-supervised learning*, dimensionality reduction dan lain lain.
- ***Anomaly Detection***. Tujuan dari *anomaly detection* adalah untuk mempelajari bagaimanakah data "normal" itu, dan menggunakannya untuk mendeteksi keadaan abnormal dari *in-*

stances, seperti item-item yang rusak pada lini produksi atau untuk menemukan trend pada sebuah *time series*.

- ***Density Estimation***. Tujuan dari *density estimation* adalah untuk melakukan estimasi *probability density function* (PDF) dari proses acak yang menghasilkan dataset tersebut. *Density estimation* biasanya digunakan untuk *anomaly detection* dengan prinsip bahwa *intances* yang berada pada daerah dengan *low density* kemungkinan besar adalah anomali. Hal ini sangat berguna juga untuk analisis data dan visualisasi data.

1 *Clustering*

Jika kita melakukan *hiking* di pegunungan, bisa jadi kita akan menemukan tanaman yang belum pernah kita lihat sebelumnya. Kemudian jika kita melihat-lihat sekitaran dan kita akan menemukan tanaman yang sama beberapa kali. Tanaman tersebut tidak betul-betul identik, tetapi mereka cukup serupa untuk kita menyatakan bahwa mereka berasal dari spesies yang sama (setidaknya genus yang sama). Kita mungkin membutuhkan seorang *botanist* untuk mengkategorikan tanaman-tanaman tersebut pada spesies apa, tetapi kita tentu tidak membutuhkan seorang ahli untuk mengidentifikasi dan mengelompokan objek-objek yang serupa ke dalam satu grup tertentu. Masalah ini disebut dengan ***clustering***, yaitu sebuah cara untuk mengidentifikasi *instance-instance* yang serupa dan mengkategorikannya ke dalam sebuah *cluster* atau grup dengan objek-objek yang sejenis.



Gambar 6.1. Perbedaan klasifikasi (kiri) dan clustering (kanan)

Seperti pada klasifikasi, setiap *instance* akan dikategorikan ke dalam sebuah grup. Tetapi, tidak seperti klasifikasi, *clustering* adalah pekerjaan *unsupervised*, seperti yang diilustrasikan pada Gambar 6.1. Pada gambar sebelah kiri adalah dataset Iris, dimana kategori spesies (*class*) dari setiap *instance* direpresentasikan dengan tanda dan warna yang berbeda, karena berasal dari dataset yang mempunyai label. Kita bisa gunakan algoritma-algoritma klasifikasi seperti regresi logistik, SVM, random forests, dll untuk memproses data-data seperti itu. Gambar sebelah kanan menunjukan data-data dari dataset yang sama tetapi tanpa label, sehingga kita tidak dapat menggunakan algoritma klasifikasi dalam kasus ini. Inilah kondisi dimana algoritma *clustering* akan berperan. Secara visual kita dengan mudah melihat bahwa data-data yang terletak sebelah kiri bawah membentuk sebuah *cluster*, tetapi tidak mudah untuk bisa menyimpulkan dari gambar sebelah kanan bahwa kumpulan data besar sebelah kanan atas terdiri dari dua *cluster* yang berbeda. Tetapi,

dataset Iris sebetulnya mempunyai *feature* lain yang bisa digunakan yaitu panjang dan lebar sepal (tidak digunakan di Gambar 6.1. Dengan memakai tambahan *feature* tersebut, algoritma *clustering* akan mampu mengidentifikasi 3 cluster yang berbeda dengan kinerja yang cukup baik (misalkan dengan menggunakan model *Gaussian Mixture*, hanya 5 data dari 150 data dikategorikan pada cluster yang salah).

Clustering dapat digunakan pada banyak aplikasi, diantaranya adalah:

- **Segmentasi Kostumer**

  Kita dapat melakukan *clustering* pada kostumer berdasarkan apa yang mereka beli, atau aktifitas pada website mereka. Hal ini penting untuk mengerti siapakah para kostumer tersebut dan apa yang mereka butuhkan, sehingga kita dapat beradaptasi dengan produk yang dibutuhkan dan membuat strategi marketing pada setiap segmen. Contoh, segmentasi kustomer sangat berguna pada *recommender systems* untuk memberikan usulan konten dimana user-user lain pada cluster yang sama sudah beli atau nikmati.

- **Analisis Data**

  Ketika kita melakukan analisa dataset baru, maka akan sangat menolong jika kita dapat mengeksekusi algoritma *clustering* dan kemudian menganalisa setiap *cluster* secara terpisah.

- **Teknik *Dimensionality Reduction***

  Ketika dataset sudah dibuat *cluster*, biasanya dimungkinkan untuk mengukur *affinity* setiap *instance* (*affinity* adalah ukuran seberapa cocok sebuah *instance* dikategorikan kedalam sebuah *cluster*). Setiap vektor *feature* sebuah *instance* kemudian dapat diganti dengan sebuah vektor yang merepresentasikan *affinity* sebuah *cluster*. Jika terdapat sejumlah k *cluster*, maka vektor tersebut berdimensi k. Vektor ini biasanya berdimensi jauh lebih rendah dibandingkan dengan vektor *feature* original, tetapi tetap bisa menyimpan cuku informasi untuk pemrosesan lebih lanjut.

- ***Anomaly Detection***

  Setiap *instance* yang memiliki *affinity* rendah terhadap semua *cluster* cenderung termasuk anomali. Contoh, jika kita sudah melakukan *clustering* terhadap user-user dari website kita berdasarkan *behaviour* mereka, kita dapat mendetksi *behaviour* yang tidak lazim atau di luar kebiasaan, misalkan jumlah *request* per detik yang tidak lazim. *Anomaly detection* sangat bermanfaat khususnya untuk deteksi *defect* pada manufaktur, atau untuk *fraud detection*.

- ***Semi-Supervised Learning***

  Jika kita hanya mempunyai beberapa label, kita bisa melakukan *clustering* dan mengasosiasikan label-label pada keseluruhan *instances* di *cluster* yang sama. Teknik ini dapat menambah banyak *instances* yang berlabel, dan kemudian digunakan untuk meningkatkan algoritma *supervised learning* dengan penambahan dataset training yang berlabel tersebut.

- ***Search Engines***

  Beberapa *search engines* dapat melakukan pencarian image-image yang serupa dengan image referensi. Untuk membangun sistem seperti itu, kita dapat menerapkan algoritma *clustering* pada semua image di dalam database, sehingga image-image yang serupa akan dikategorikan ke dalam *cluster* yang sama. Kemudian, jika seorang user menyediakan image referensi, yang perlu kita lakukan adalah menggunakan model *cluster* yang telah ditraining

tadi untuk menemukan *cluster* dari image referensi. Selanjutnya kita dapat mengeluarkan image-image yang serupa dari *cluster* tersebut yang memang serupa dengan image referensi.

- **Segmentasi Image**

  Dengan melakukan *clustering* piksel-piksel berdasarkan wana, kemudian mengganti setiap warna piksel-piksel dengan warna rata-rata sebuah *cluster*, maka hal ini memungkinkan kita untuk mengurangi jumlah warna-warna yang berbeda dalam sebuah image. Segementasi image banyak digunakan pada sistem deteksi dan pelacakan (*tracking*) objek, karena lebih mudah untuk mendeteksi kontour (hasil segementasi warna) dari setiap objek.

Sebagai catatan, tidak ada definisi universal menyangkut sebuah *cluster*. Definisi ini akan sangat tergantung pada konteks, dan algoritma yang berbeda akan menangkap (*capture*) jenis *cluster* yang berbeda pula. Beberapa algoritma mencari *instance-instance* di sekitar titik tertentu yang disebut *centroid*. Algoritma lain mencari daerah kontinyu yang dibentuk berdasarkan kepadatan lokasi dari *instance-instance*. Jenis *cluster* terakhir ini bisa mempunyai bentuk macam-macam. Beberapa algoritma bekerja secara berjenjang (*hierarchical*), mencari *cluster* dari *cluster-cluster*. Dan banyak lagi algoritma lain.

2 *Clustering* dengan *K-Means*

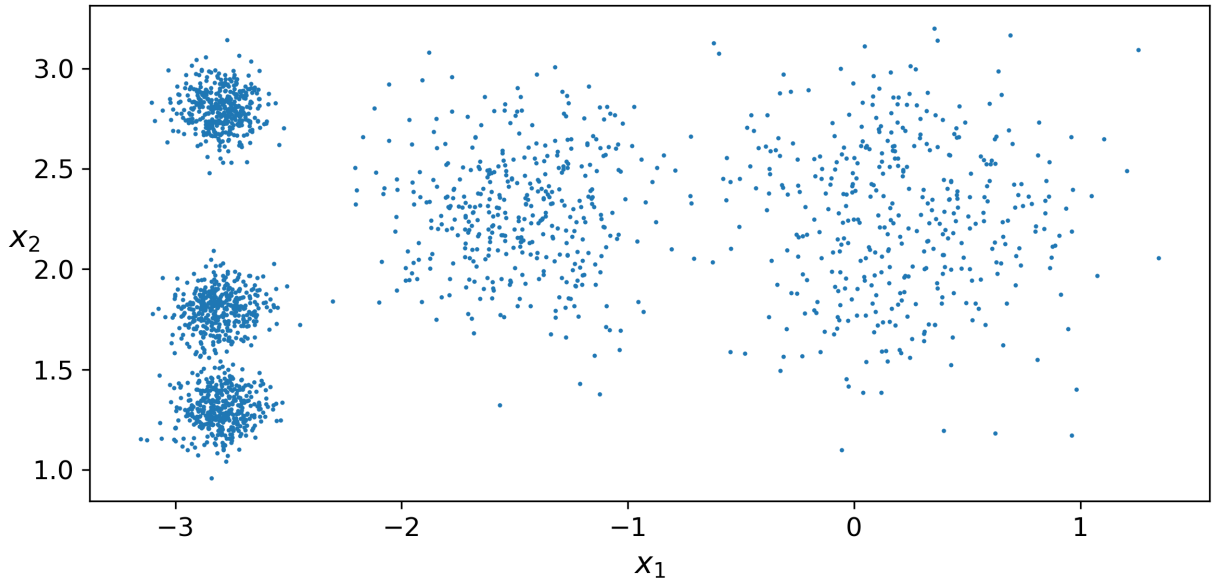
Kita akan melihat dua jenis algoritma *clustering* yang paling populer, yaitu **K-Means** dan **DBSCAN**. Kita juga akan mengeksplorasi aplikasi kedua algoritma tersebut, seperti *dimensionality reduction* nonlinier, *semi-supervised learning* dan *anomaly detection*.

Jika terdapat dataset yang tidak seimbang seperti yang ditampilkan pada Gambar 6.2, dimana terdapat 5 gumpalan (*blobs*) dari *instances*. Algoritma ***K-Means*** merupakan algoritma sederhana yang mampu melakukan *clustering* untuk dataset seperti ini secara cepat dan efisien. Seringkali hasil yang diperoleh sangat baik meskipun hanya menggunakan sedikit iterasi. Algoritma ini diajukan oleh Stuart Lloyd dari Bell Labs pada tahun 1957 sebagai teknik *pulse-code modulation*, tetapi baru dipublikasikan di luar perusahaan pada tahun 1982. Pada tahun 1965, Edward W. Forgy mempublikasikan algoritma yang sebetulnya sama dengan *K-Means*, sehingga *K-Means* juga terkenal dengan nama Lloyd-Forgy.

Sekarang kita akan coba melakukan training pengkluster untuk *K-Means* pada dataset di atas. Algortima ini akan mencoba mencari pusat dari setiap gumpalan dan melakukan pengasosiasian setiap *data instance* pada gumpalan terdekat.

- Berikut adalah cara membangkitkan data seperti yang ditunjukan pada Gambar 6.2

[2]:
```
from sklearn.datasets import make_blobs
import numpy as np

blob_centers = np.array(
    [[ 0.2,  2.3],
     [-1.5 ,  2.3],
     [-2.8,  1.8],
     [-2.8,  2.8],
     [-2.8,  1.3]])
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])
```



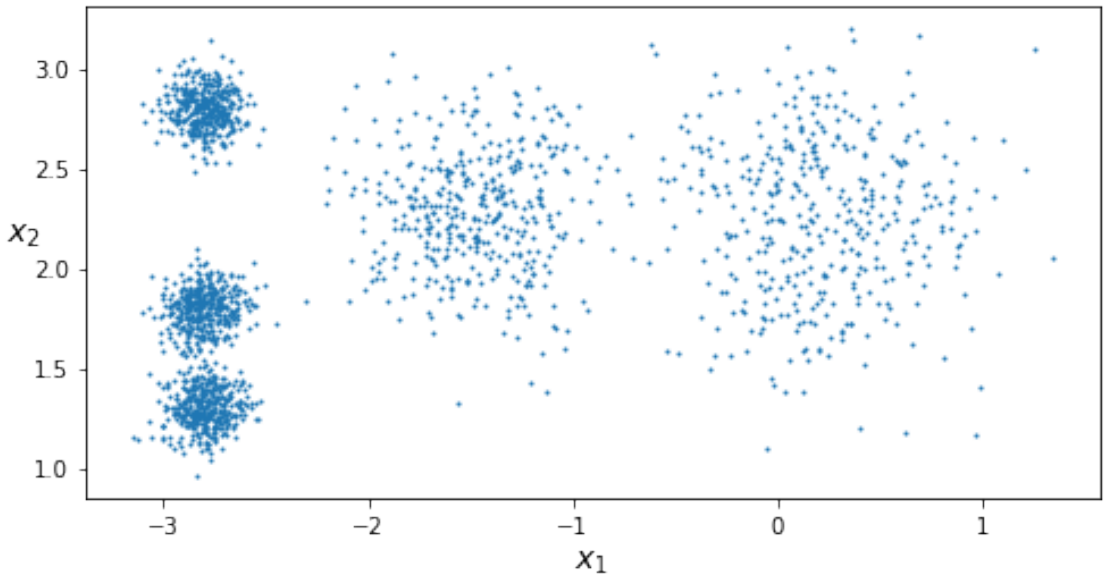
Gambar 6.2. Dataset yang terdiri dari 5 gumpal data dan tanpa label

```
X, y = make_blobs(n_samples=2000, centers=blob_centers,
                  cluster_std=blob_std, random_state=7)
```

Kemudian untuk melakukan plotting dataset

[3]:
```
import matplotlib.pyplot as plt
def plot_clusters(X, y=None):
    plt.scatter(X[:, 0], X[:, 1], c=y, s=1)
    plt.xlabel("$x_1$", fontsize=14)
    plt.ylabel("$x_2$", fontsize=14, rotation=0)
```

[4]:
```
plt.figure(figsize=(8, 4))
plot_clusters(X)
plt.show()
```



- Sekarang kita akan coba melakukan training algoritma *clustering* dari *K-Means* pada dataset di atas. Algortima ini akan mencoba mencari pusat dari setiap gumpalan dan melakukan pengasosiasian setiap *data instance* pada gumpalan terdekat.

[5]:
```
from sklearn.cluster import KMeans
```

[6]:
```
k = 5
kmeans = KMeans(n_clusters=k, random_state=42)
y_pred = kmeans.fit_predict(X)
```

Kita harus menspesifikasikan jumlah kluster *k* yang harus ditemukan oleh algoritma. Pada gambar ini sudah sangat jelas bahwa jumlah *cluster* yang harus ditemukan adalah `k=5`. Tetapi secara umum tidak akan semudah ini. Setiap *instance* akan diasosiasikan ke salah satu dari 5 *cluster* tadi. Pada konteks *clustering*, label dari setiap *data instance* adalah indeks *cluster* yang ditentukan oleh hasil eksekusi algoritma (beda dengan label pada konsep klasifikasi di *supervised learning*). *Instance* dari `KMeans` menyimpan copy dari label-label *instances* hasil training pada variabel `labels_instance`.

[7]:
```
y_pred
```

[7]:
```
array([4, 1, 0, ..., 3, 0, 1], dtype=int32)
```

[8]:
```
y_pred is kmeans.labels_
```

[8]:
```
True
```

Kita juga dapat melihat hasil estimasi 5 lokasi titik pusat (*centroids*) dari setiap *cluster* yang ditemukan:

[9]:
```
kmeans.cluster_centers_
```

[9]:
```
array([[ 0.20876306,  2.25551336],
       [-2.80389616,  1.80117999],
       [-1.46679593,  2.28585348],
       [-2.79290307,  2.79641063],
       [-2.80037642,  1.30082566]])
```

Dimana pada array di atas, baris pertama adalah centroid dari *cluster* 0, baris kedua adalah centroid dari *cluster* 1, dan selanjutnya.

Kita dapat menentukan dengan mudah termasuk *cluster* mana sebuah data *instance* baru, yaitu menggunakan jarak centroid terdekat dari data tersebut.

[10]:
```
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
kmeans.predict(X_new)
```

[10]:
```
array([0, 0, 3, 3], dtype=int32)
```

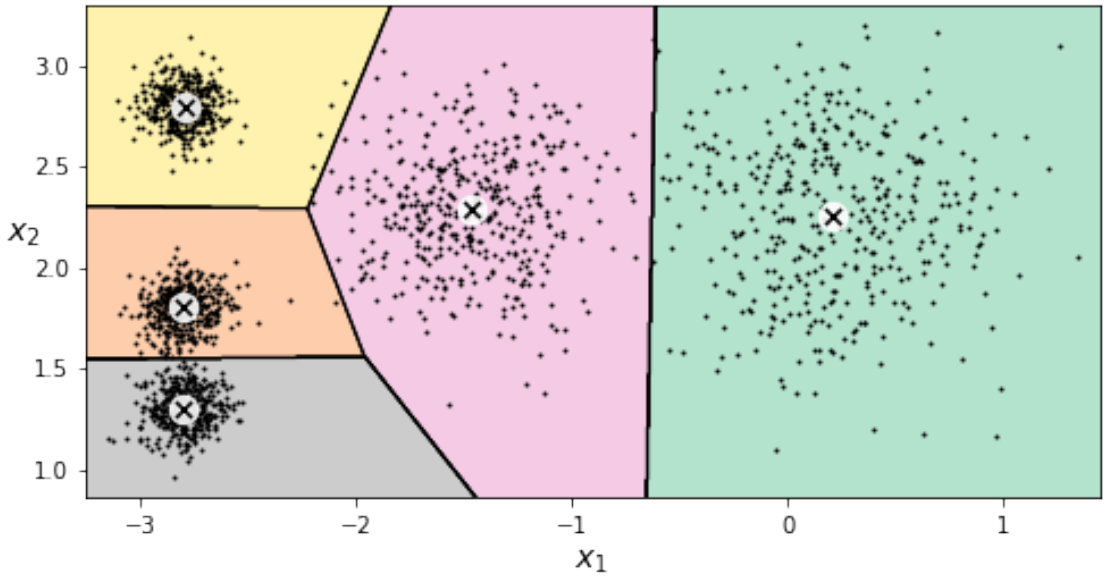
Terlihat bahwa data [0,2] dan [3,2] termasuk *cluster* 0, sedangkan [-3,3] dan [-3, 2.5] termasuk *cluster* 3. Hal ini dapat kita verifikasi sendiri secara visual pada gambar di atas.

Kita juga dapat memplot batas keputusan (*decision boundary*) dari *cluster-cluster* tersebut dengan menggunakan diagram Voronoi (*Voronoi tesselation*)

[11]:
```
def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

def plot_centroids(centroids, weights=None, circle_color='w', cross_color='k'):
    if weights is not None:
        centroids = centroids[weights > weights.max() / 10]
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='o', s=30, linewidths=8,
                color=circle_color, zorder=10, alpha=0.9)
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='x', s=50, linewidths=50,
                color=cross_color, zorder=11, alpha=1)

def plot_decision_boundaries(clusterer, X, resolution=1000, show_centroids=True,
                             show_xlabels=True, show_ylabels=True):
    mins = X.min(axis=0) - 0.1
    maxs = X.max(axis=0) + 0.1
    xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], resolution),
                         np.linspace(mins[1], maxs[1], resolution))
    Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                cmap="Pastel2")
```

```
    plt.contour(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                linewidths=1, colors='k')
    plot_data(X)
    if show_centroids:
        plot_centroids(clusterer.cluster_centers_)

    if show_xlabels:
        plt.xlabel("$x_1$", fontsize=14)
    else:
        plt.tick_params(labelbottom=False)
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
        plt.tick_params(labelleft=False)
```

[12]:
```
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans, X)
plt.show()
```



Pada gambar di atas lokasi setiap centroid dilambangkan dengan tanda 'X'. Sebagian besar dari *data instances* di*assign* pada *cluster* yang tepat, tetapi bisa jadi beberapa *instances* di*assign cluster* yang salah (terutama pada batas keputusan antara *cluster* pada kiri atas dengan *cluster* di tengah. Sebetulnya, algoritma *K-Means* tidak mempunyai kinerja yang cukup baik jika gumpalan-gumpalan data mempunyai ukuran diameter yang tidak sama. Hal ini karena yang diperdulikan oleh algoritma *K-Means* saat melakukan *assigning instance* terhadap *cluster* hanya jaraknya ke *centroid*.

Cara melakukan *assigning* setiap *instance* terhadap *cluster* dapat dibagi dua yaitu *hard clustering*

dan soft clustering. *Jika setiap* instance* hanya di*assign* pada satu *cluster* saja maka disebut dengan *Hard clustering*. Sedangkan pada *soft clustering*, setiap *instance* akan diberikan *score* pada setiap *cluster*. *Score* boleh jadi merupakan jarak antara *instance* dan centroid, atau bisa jadi dalam bentuk *similarity score* (*affinity*), seperti Gaussian Radial Basis Function (RBF) yang telah dijelaskan di *Chapter 5*.

2.1 Algoritma *K-Means*

Setelah prinsip *K-Means* dijelaskan di atas, bagaimanakan sebetulnya algoritma *K-Means* bekerja? Jika dimisalkan kita diberikan centroid-centroid, kita bisa melabeli *instances* dengan cara *assign* masing-masing *instance* pada *cluster* dengan centroid terdekat. Sebaliknya, jika semua *instance* diberikan label, kita dapat dengan mudah mencari lokasi semua centroid dengan menghitung *mean* dari semua *instances* pada setiap *cluster*. Tetapi, kenyataannya kita tidak diberikan informasi keduanya. Oleh sebab itu, solusi yang paling logis adalah dengan menempatkan centroid secara acak, yaitu dengan memilih sejumlah *k instances* secara random, dan menggunakan semua lokasinya sebagai centroid awal. Kemudian dilanjutkan dengan melabeli setiap *instances*, *update* centroid lagi, labeli setiap *instances* lagi, demikian seterusnya, sampai lokasi centroid-centroid tersebut tidak berubah lagi. Algoritma dijamin akan konvergen dengan jumlah iterasi terbatas (biasanya cukup kecil), dan tidak akan berosilasi selamanya. Secara singkat algoritma *K-Means* bekerja dengan langkah sebagai berikut:

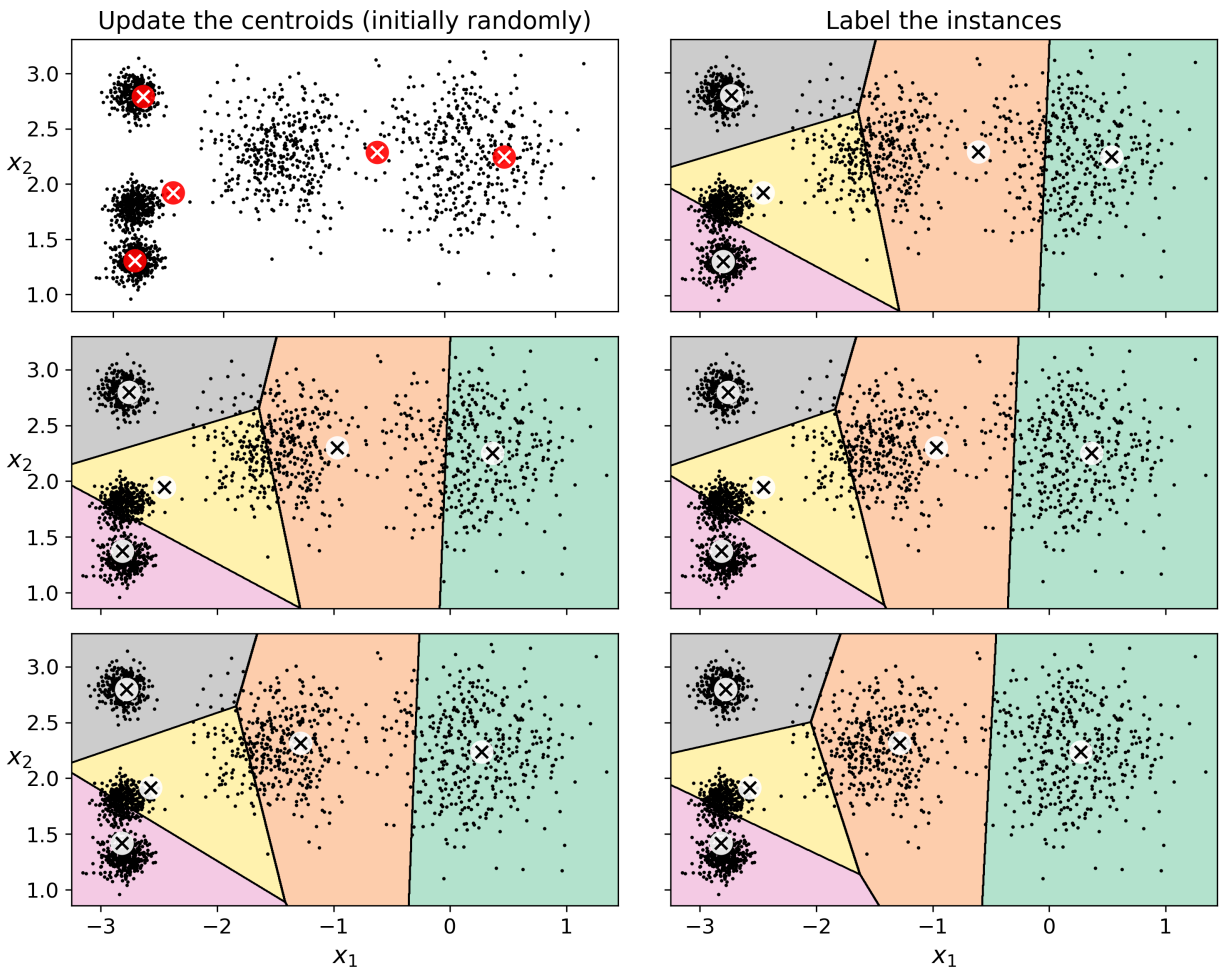
- Inisialisasi sebanyak *k* centroid secara random, biasanya dipilih secara acak dari dataset, dan kemudian ditempatkan pada masing-masing lokasi
- *Assign* setiap *instance* pada centroid terdekat
- Update masing-masing centroid yang merupakan rata-rata *instance-instance* yang di*assign* pad centroid sebelumnya
- Ulangi dua prosedur di atas sampai konvergen (sampai centroid-centroid tersebut tidak bergerak)

Catatan. Pengukuran jarak antara centroid dan *instance* bisa menggunakan *Euclidean Distance*.

Gambar 6.3 menunjukan centroid-centroid diinisialisasi secara random (kiri atas), kemudian setiap *instances* dilabeli (kanan atas), kemudian setiap centroid diupdate lagi (kiri tengah), *instances* dilabeli kembali (kanan tengah) dan terus selanjutnya. Seperti yang ditujukkan Gambar 6.3 hanya dengan tiga kali iterasi maka akan diperoleh *cluster-cluster* yang mendekati optimal.

Catatan. Tingkat kompleksitas dari algoritma *K-Means* umumnya linier terhadap jumlah *instance* m, jumlah *cluster* k, dan jumlah dimensi (*feature*) n. Hal ini terjadi ketika data mempunyai struktur *cluster*. Jika tida ada struktur *cluster*, pada kasus terburuk maka kompleksitas akan naik secara eksponensial dengan bertambahnya jumlah *instance*. Di ranah praktis, hal ini jarang terjadi, sehingga *K-Means* secara umum adalah salah satu algoritma *clustering* yang tercepat.

Kendatipun algoritma ini dijamin akan konvergen, tetapi bisa jadi tidak konvergen pada solusi yang tepat (optimal), atau hanya konvergen ke optimum lokal (suboptimal). Konvergensi ke solusi yang optimal atau suboptimal akan tergantung pada inisialisasi letak centroid secara acak. Gambar 6.4 menunjukan kondisi dimana algoritma konvergen hanya pada solusi suboptimal, jika kita tidak beruntung saat memilih lokasi centroid pertama kali secara acak. Cara-cara untuk mengurangi konvergensi yang tidak optimal akibat inisialiasi centroid akan dijelaskan pada bagian selanjutnya.



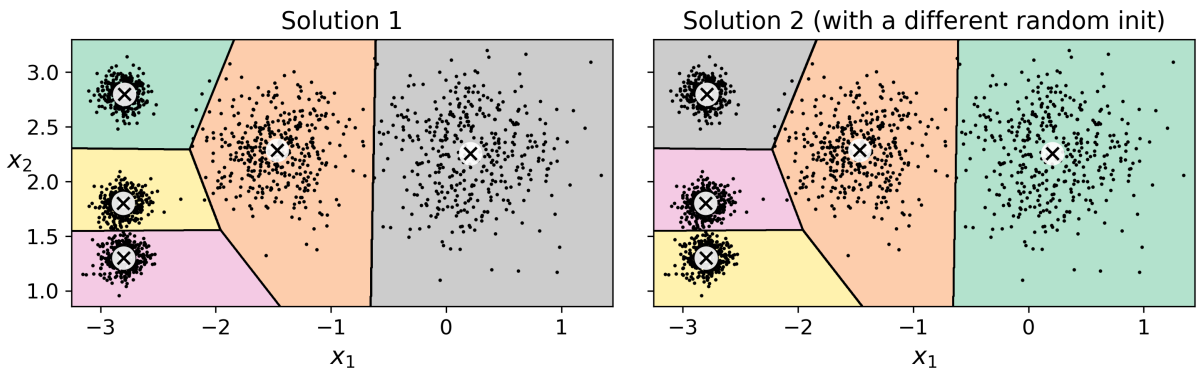
Gambar 6.3. Ilustrasi algoritma K-Means

2.2 Metode Inisialisasi Centroid

Jika kita bisa mengira-ngira dimana seharusnya centroid-centroid berada (misalkan setelah mengeksekusi algoritma *clustering* sebelumnya), maka kita dapat melakukan *setting* hyperparameter `init` dengan sebuah *NumPy array* yang berisi centroid-centroid perkiraan tersebut dan jadikan `n_init = 1`, seperti contoh bagian program berikut.

```
>>> good_init = np.array([[-3,3],[-3,2],[-3,1],[-1,2],[0,2]])
>>> kmeans = KMeans(n_clusters=5,init=good_init, n_init=1)
```

Solusi lain adalah dengan mengeksekusi algorita beberapa kali dengan inisialisasi acak yang berbeda-beda dan pilihlah solusi terbaik. Jumlah inisialiasi random dikontrol dengan hyperparameter `n_init`. Secara default, hyperparameter ini di*set* sama dengan 10, yang artinya algoritma *K-Means* akan dieksekusi 10 kali ketika fungsi `fit()` dipanggil, dan `Scikit Learn` akan memilihkan solusi terbaik dari 10 eksekusi tersebut. Metrik yang digunakan oleh `Scikit Learn` untuk memilih solusi terbaik adalah ***model's inertia***, yang dapat didefinisikan sebagai jarak kuadrat rata-rata (*mean squared distance*) antara setiap *instance* terhadap centroid terdekat. Sebagai ilustrasi, pada Gambar 6.4 harga dari *model's inertia* adalah sekitar 223,3 untuk sebelah kiri dan 237,5 untuk sebelah kanan. Sedangkan untuk gambar diagram Voronoi pertama hasil eksekusi program



Gambar 6.4. Solusi suboptimal karena ketidakberuntungan inisialisasi lokasi centroid-centroid secara acak

di *model's inertia* bernilai 211,6. *Class* `KMeans` dari `Scikit Learn` akan melakukan eksekusi algorima sebanyak `n_init` kali dan memilih model dengan *inertia* terendah. Pada contoh ini, diagram Voronoi pertama akan dipilih. *Model's inertia* dapat diakses melalui variabel `inertia_instance` seperti perintah berikut.

[13]:
```
kmeans.inertia_
```

[13]:
```
211.5985372581684
```

Metode `score()` akan menghasilkan *inertia* yang negatif karena metode `score()` dari prediktor harus mengikuti aturan `Scikit Learn` yaitu *'greater is better'*. Artinya, jika sebuah prediktor dikatakan lebih baik dibandingkan yang lain, maka metode `score()` dari prediktor tersebut harus menghasilkan *score* yang paling besar.

[14]:
```
kmeans.score(X)
```

[14]:
```
-211.59853725816856
```

Modifikasi yang paling penting untuk memperbaiki kinerja dari algoritma *K-Means* salah satunya adalah algoritma ***K-Means++***, yang diajukan pada tahun 2006 oleh David Arthur dan Sergei Vassilvitskii. Mereka mengajukan inisialisasi yang lebih cerdas, yaitu dengan cara memilih centroid-centroid yang diatur berjauhan satu sama lain. Modifikasi ini membuat algoritma *K-Means* mempunyai kemungkinan yang kecil untuk konvergen ke solusi suboptimal. Mereka membuktikan bahwa meskipun dibutuhkan penambahan komputasi untuk inisialisasi yang lebih cerdas, tetapi modifikasi tersebut dapat mengurangi secara drastis jumlah eksekusi algoritma yang dibutuhkan untuk menemukan solusi optimal. Berikut adalah algoritma inisialisasi *K-Means++*.

1. Ambil satu centroid $c^{(1)}$ yang dipilih secara acak dari dataset.
2. Ambil centroid baru $c^{(i)}$, yang dipilih berdasarkan *instance* $x^{(i)}$ yang memiliki probabilitas tertinggi dimana probabilitas didefinisikan sebagai $\frac{D\left(x^{(i)}\right)^2}{\sum_{j=1}^{m} D\left(x^{(j)}\right)^2}$ dan $D\left(x^{(i)}\right)$ adalah jarak instance $x^{(i)}$ terhadap centroid terdekat yang telah dipilih sebelumnya. Distribusi probabilitas ini menjamin *instance-instance* dengan jarak terjauh dari centroid-centroid yang telah

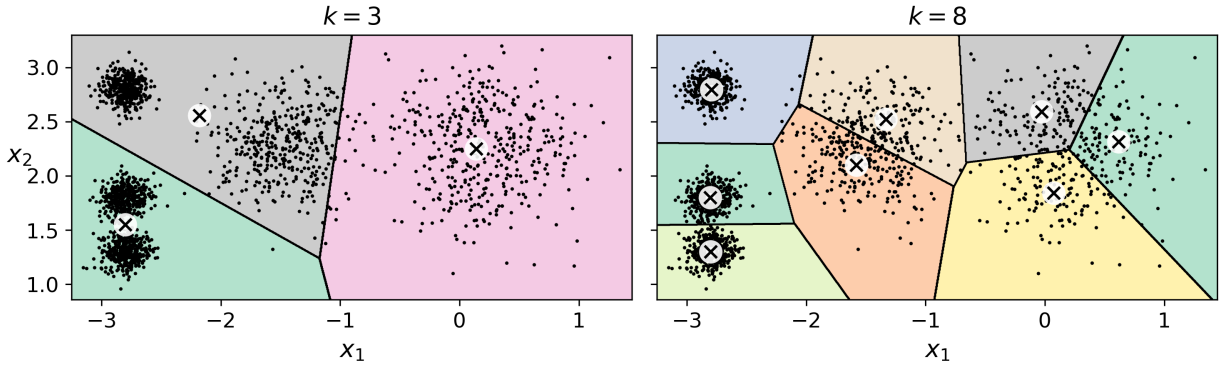
dipilih akan lebih mungkin dipilih kemudian menjadi centroid-centroid selanjutnya.
3. Proses sebelumnya berulang sampai semua centroid yang berjumlah k sudah terpilih semuanya.

Class `KMeans` menggunakan metode inisialisasi di atas sebagai default. Jika ingin memaksanya untuk menggunakan metode original (memilih sejumlah k *instances* secara acak sebagai centroid-centroid inisial), maka dapat digunakan hyperparameter `init = "random"`. Tetapi kemungkinannya kita tidak akan pernah memakainya, dengan alasan seperti penjelasan di atas.

Beberapa metode untuk mempercepat algoritma *K-Means*, bisa dilihat pada paper Charles Elkan 2003 dan David Sculley 2010.

2.3 Menemukan Jumlah *Cluster* yang Optimum

Sejauh ini kita telah menggunakan jumlah *cluster* sebanyak $k = 5$ karena bisa diobservasi secara langsung dari data, yang memang kenyataannya berjumlah 5. Tetapi secara umum, tidak selalu mudah untuk menentukan jumlah *cluster* k dan hasilnya akan sangat buruk ketika kurang tepat menentukan jumlah *cluster* k. Seperti yang terlihat pada Gambar 6.6, jika $k = 3$ atau 8 model yang dihasilkan terlihat buruk.



Gambar 6.5. Ilustrasi pemilihan yang tidak tepat dalam jumlah cluster, jika k terlalu kecil maka cluster-cluster yang seharusnya terpisah akan tergabung (kiri) dan jika k terlalu besar beberapa cluster yang harusnya satu menjadi terpisah-pisah

Gambar 6.6 dihasilkan dengan kode program berikut.

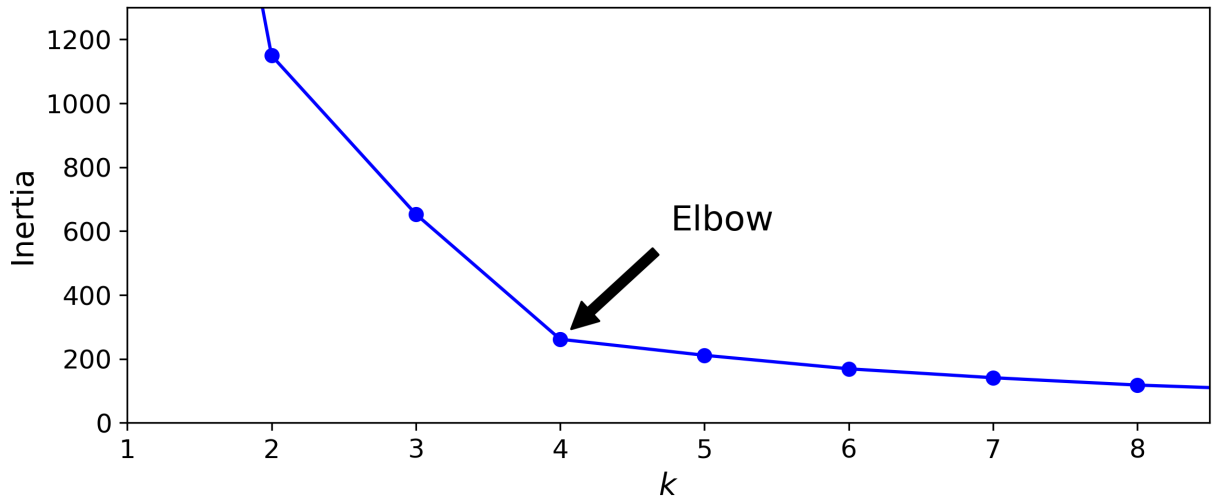
[15]:
```
kmeans_k3 = KMeans(n_clusters=3, random_state=42)
kmeans_k8 = KMeans(n_clusters=8, random_state=42)
```

[16]:
```
def plot_clusterer_comparison(clusterer1, clusterer2, X, title1=None,␣
↪title2=None):
    clusterer1.fit(X)
    clusterer2.fit(X)

    plt.figure(figsize=(10, 3.2))

    plt.subplot(121)
    plot_decision_boundaries(clusterer1, X)
```

```
    if title1:
        plt.title(title1, fontsize=14)

    plt.subplot(122)
    plot_decision_boundaries(clusterer2, X, show_ylabels=False)
    if title2:
        plt.title(title2, fontsize=14)

# Jika ingin menampilkan gambar aktifkan dua baris di bawah ini
#plot_clusterer_comparison(kmeans_k3, kmeans_k8, X, "$k=3$", "$k=8$")
#plt.show()
```

Dalam menentukan jumlah *cluster*, parameter *inertia* bukan merupakan indikator yang baik karena *inertia* akan mengecil ketika kita menambah jumlah k. Jika jumlah *cluster* bertambah, maka akan makin dekat setiap *intance* ke masing-masing *centroid* sehingga *inertia* akan semakin mengecil pula. Pada Gambar 6.5, untuk $k = 3$ *inertia* adalah 653,2 dan untuk $k = 8$ *inertia* bernilai lebih kecil yaitu 119.1.

Gambar 6.6 menunjukan gambar *inertia* sebagai fungsi dari jumlah *cluster* k. Terlihat bahwa *inertia* akan menurun sangat tajam ketika k membesar sampai 4, kemudian menurun lebih lambat ketika k diperbesar lebih dari 4. Apabila kita tidak tahu lebih jauh, maka 4 merupakan pilihan yang sangat beralasan. Pilihan lebih kecil 4 akan terlalu dramatis sedangkan terlalu besar tidak akan terlalu banyak menolong dan bisa jadi malah memotong sebuah *cluster* yang seharusnya menjadi beberapa bagian (setengahnya).

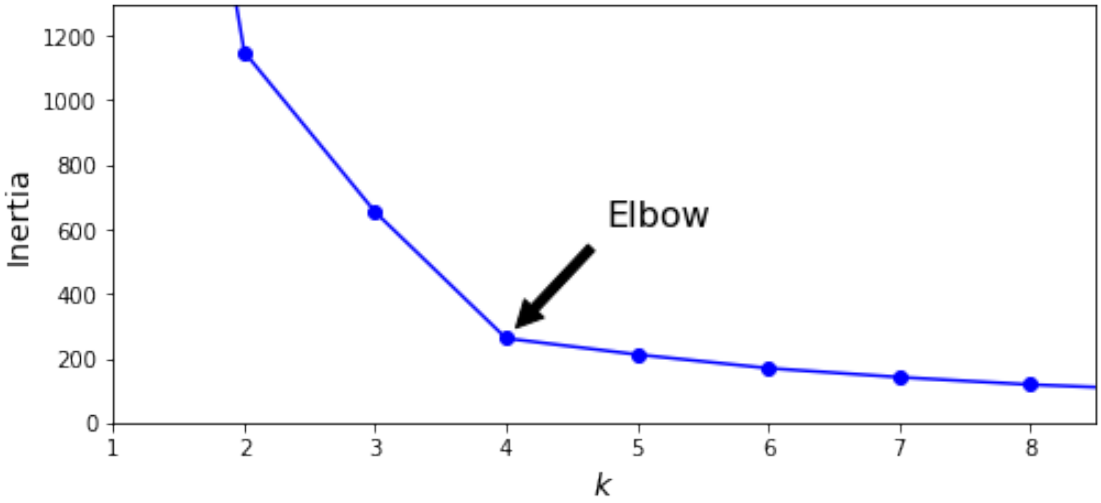


Gambar 6.6. Inertia sebagai fungsi dari jumlah cluster k, kurva biasanya menunjukan titik perubahan (infleksi) seperti siku (elbow)

Gambar 6.6 diperoleh dari program berikut.

[17]:
```
kmeans_per_k = [KMeans(n_clusters=k, random_state=42).fit(X)
                for k in range(1, 10)]
inertias = [model.inertia_ for model in kmeans_per_k]
```

[18]:
```
plt.figure(figsize=(8, 3.5))
plt.plot(range(1, 10), inertias, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Inertia", fontsize=14)
plt.annotate('Elbow',
             xy=(4, inertias[3]),
             xytext=(0.55, 0.55),
             textcoords='figure fraction',
             fontsize=16,
             arrowprops=dict(facecolor='black', shrink=0.1)
            )

plt.axis([1, 8.5, 0, 1300])
plt.show()
```



Teknik untuk menentukan harga terbaik dari jumlah *cluster* biasanya bersifat kasar (tidak presisi). Pendekatan yang lebih presisi tetapi mahal secara perhitungan (*computationally expensive*) adalah *silhouette score*, yang merupakan rata-rata koefisien *silhouette* dari semua *instance*. *Instance silhouette* dihitung berdasarkan Persamaan (6.1).

Persamaan (6.1) *Silhouette coefficient* sebuah *instance*

$$S_C = \frac{b - a}{\max(a, b)}$$

dimana a adalah rata-rata jarak ke *instance-instance* lain pada *cluster* yang sama (rata-rata jarak *intra-cluster*) dan b menyatakan rata-rata jarak ke *cluster* terdekat atau juga berarti rata-rata jarak ke *instance-instance* yang berada pada *cluster* terdekat (meminimalkan b)

Harga koefisien S_C bervariasi dari -1 s/d $+1$. Koefisien dengan harga mendekati $+1$ artinya *instance* sudah berada pada *cluster* yang betul, harga 0 berarti *instance* dekat ke perbatasan *cluster*,

sedangkan -1 berarti *instance* tersebut mendapatkan penempatan *cluster* yang salah.

Untuk menentukan S_C, kita bisa gunakan fungsi `silhoutte_score()` pada `Scikit-Learn`.

[19]:
```
from sklearn.metrics import silhouette_score
```

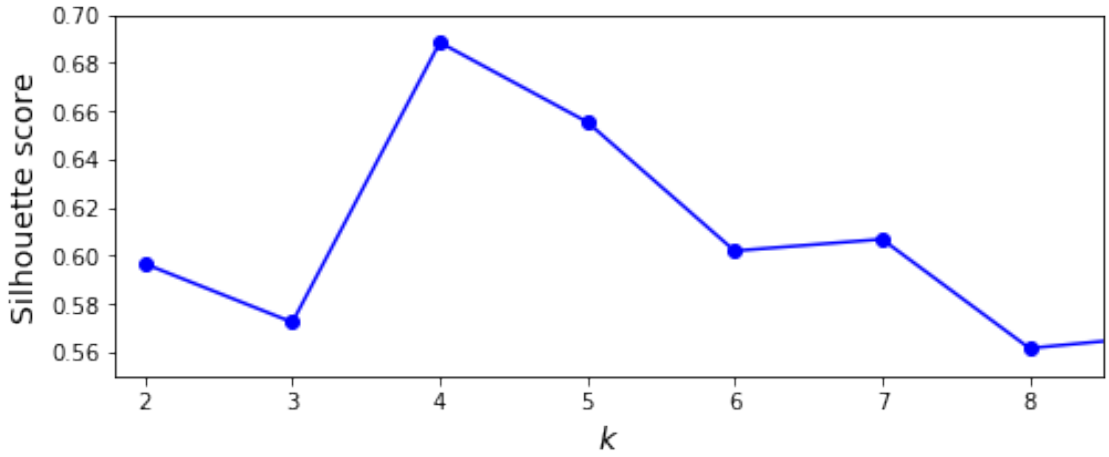
[20]:
```
silhouette_score(X, kmeans.labels_)
```

[20]:
```
0.655517642572828
```

Program berikut merupakan harga S_C sebagai fungsi dari jumlah *cluster*.

[21]:
```
silhouette_scores = [silhouette_score(X, model.labels_)
                     for model in kmeans_per_k[1:]]
```

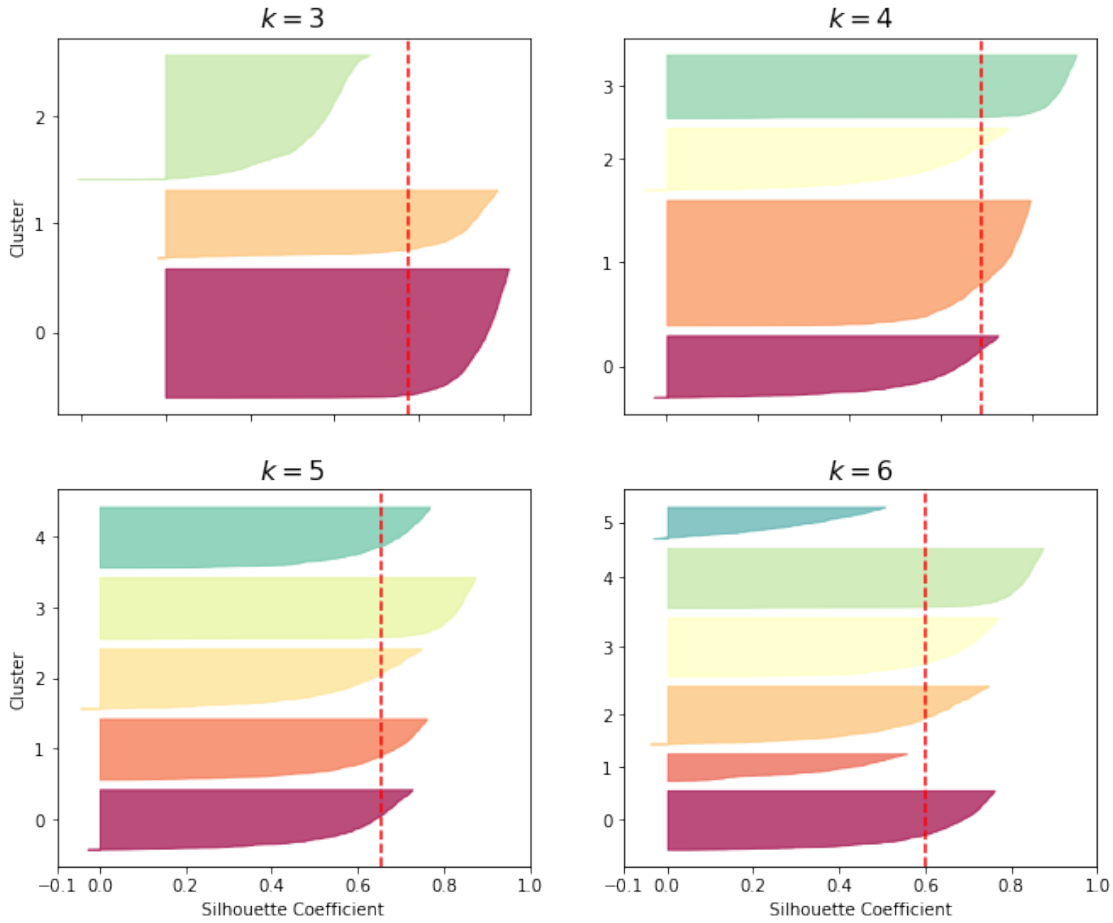
[22]:
```
plt.figure(figsize=(8, 3))
plt.plot(range(2, 10), silhouette_scores, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Silhouette score", fontsize=14)
plt.axis([1.8, 8.5, 0.55, 0.7])
plt.show()
```



Terlihat pada gambar di atas, visualisasinya lebih baik dari Gambar 6.6 yang menggunakan *inertia*. Meskipun mengkonfirmasi bahwa $k = 4$ adalah pilihan yang baik, tetapi $k = 5$ juga merupakan alternatif pilihan yang baik, jauh lebih baik dibandingkan 6 dan 7. Hal semacam ini tidak terlihat ketika menggunakan *inertia* pada Gambar 6.6.

Visualisasi yang lebih informatif dapat diperoleh ketika kita membuat gambar setiap koefisien *Silhouette*, diurutkan berdasarkan *cluster* yang di*assign* dan berdasarkan harga S_C. Visualisasi ini disebut dengan diagram *Silhoutte*. Pemilihan jumlah *cluster* dapat dilakukan berdasarkan analisis menggunakan diagram *Silhoutte* yang dapat dilihat pada link berikut Silhoutte Analysis. Kode program berikut adalah contoh untuk menampilkan diagram *silhouette* dengan jumlah *cluster* k bervariasi.

[26]:
```
from sklearn.metrics import silhouette_samples
from matplotlib.ticker import FixedLocator, FixedFormatter
import matplotlib as mpl

plt.figure(figsize=(11, 9))

for k in (3, 4, 5, 6):
    plt.subplot(2, 2, k - 2)

    y_pred = kmeans_per_k[k - 1].labels_
    silhouette_coefficients = silhouette_samples(X, y_pred)

    padding = len(X) // 30
    pos = padding
    ticks = []
    for i in range(k):
        coeffs = silhouette_coefficients[y_pred == i]
        coeffs.sort()

        color = mpl.cm.Spectral(i / k)
        plt.fill_betweenx(np.arange(pos, pos + len(coeffs)), 0, coeffs,
                          facecolor=color, edgecolor=color, alpha=0.7)
        ticks.append(pos + len(coeffs) // 2)
        pos += len(coeffs) + padding

    plt.gca().yaxis.set_major_locator(FixedLocator(ticks))
    plt.gca().yaxis.set_major_formatter(FixedFormatter(range(k)))
    if k in (3, 5):
        plt.ylabel("Cluster")

    if k in (5, 6):
        plt.gca().set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
        plt.xlabel("Silhouette Coefficient")
    else:
        plt.tick_params(labelbottom=False)

    plt.axvline(x=silhouette_scores[k - 2], color="red", linestyle="--")
    plt.title("$k={}$".format(k), fontsize=16)
plt.show()
```

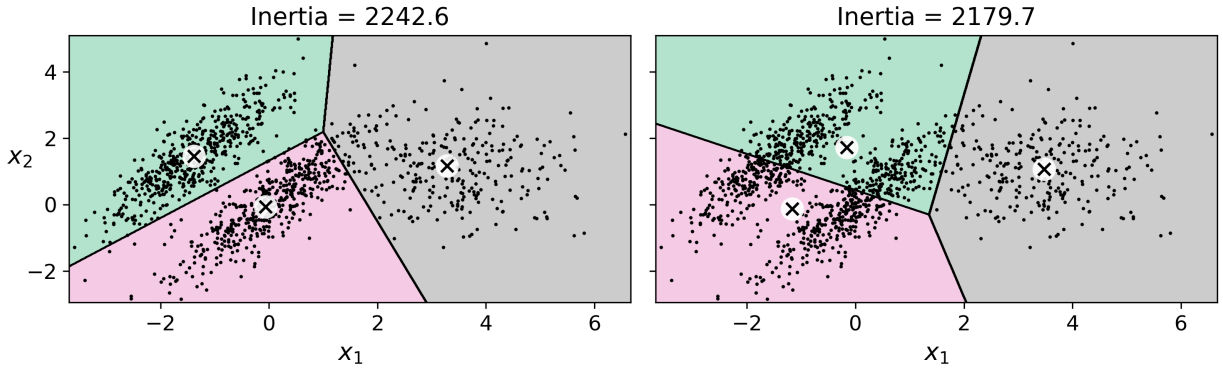


Setiap diagram berisi grafik berbentuk pisau untuk tiap *cluster*. Tinggi grafik pisau menyatakan jumlah *instance* yang dimiliki *cluster*, sedangkan lebarnya menyatakan S_C terurut dari *instance-instance* pada *cluster* (lebih lebar lebih baik). Garis putus-putus vertikal pada gambar di atas menyatakan skor *silhouette* untuk setiap jumlah *cluster* k yang berbeda. Ketika mayoritas *instance* yang berada pada satu *cluster* memiliki koefisien yang lebih kecil dari skor ini, maka dapat disimpulkan bahwa *cluster* yang diperoleh adalah buruk karena menandakan *instance-instance* tersebut terlalu dekat terhadap *cluster-cluster* lain. Tetapi ketika $k = 4$ atau $k = 5$, *cluster-cluster* tersebut terlihat lebih baik karena mayoritas *instance* memiliki koefisien yang lebih besar (sebelah kanan) dari garis putus-putus merah dan mendekati angka 1. Ketika $k = 4$, indeks *cluster* 1 (ketiga dari atas) terlihat mempunyai ukuran besar. Ketika $k = 5$, semua *cluster* terlihat mempunyai ukuran yang sama. Sehingga meskipun keseluruhan skor *silhouette* dari $k = 4$ terlihat lebih baik dibandingkan saat $k = 5$, tetapi memilih $k = 5$ merupakan ide yang baik juga untuk mendapatkan ukuran *cluster* yang hampir sama.

2.4 Keterbatasan dari *K-Means*

Kendatipun banyak kelebihan dari *K-Means*, diantaranya yang paling utama adalah kecepatan dan kemudahan untuk penskalaan, *K-Means* bukan merupakana algoritma yang sempurna tanpa kelemahan. Seperti yang telah dijelaskan sebelumnya, kita harus mengeksekusi algortima beber-

apa kali untuk menghindari solusi suboptimal, ditambah kita harus menspesifikasikan jumlah *cluster* sebelumnya yang cukup merepotkan. Selain itu *K-Means* tidak berperilaku baik ketika ukuran *cluster* bervariasi, kerapatan berbeda atau bentuk yang tidak sferis (bundar). Sebagai contoh Gambar 6.7 menunjukan bagaimana *K-Means* tidak dapat melakukan *clustering* dengan baik ketika dataset mempunyai *cluster-cluster* berbentuk elips dengan ukuran, kerapatan dan orientasi yang berbeda-beda.



Gambar 6.7. Ilustrasi K-Means tidak bisa melakukan clustering secara baik untuk data blobs yang berbentuk elips

Dapat kita lihat di Gambar 6.7, tidak ada solusi yang baik di kedua gambar di atas. Solusi pada gambar sebelah kiri lebih baik, tetapi masih memotong 25% *instance-instance* yang seharusnya berada di *cluster* tengah menjadi dikategorikan pada *cluster* sebelah kanan. Solusi pada gambar sebelah kanan sangat buruk, meskipun mempunyai *inertia* yang lebih rendah. Sehingga, keberhasilan algoritma *clustering* bergantung pada bentuk data. Pada jenis data dengan *cluster* berbentuk elips, model *Gaussian mixture* akan bekerja sangat baik.

Catatan. Untuk menghindari hasil yang buruk, sangat penting untuk melakukan penskalaan *feature* sebelum mengeksekusi algoritma *K-Means*. Penskalaan *cluster* tidak akan menjamin semua *cluster* terlihat baik dan sferis, tetapi setidaknya akan memperbaiki beberapa hal.

3 Contoh Penggunaan *Clustering*

Pada bagian ini akan ditunjukan beberapa cara penggunaan *clustering*. Pada bahasan ini, kita akan menggunakan algoritma *K-Means*, tetapi kita bisa juga menggunakan algoritma-algoritma *clustering* lain (silahkan dicoba sendiri).

3.1 Penggunaan *Clustering* untuk Segmentasi Image (*Image Segmentation*)

Segementasi image adalah cara untuk mempartisi image ke dalam beberapa segmen. Pada **Segmentasi Semantik**, semua piksel-piksel yang merupakan bagian dari objek yang sama akan di*assign* pada segmen yang sama juga. Sebagai contoh, pada sistem vision dari *self-driving car*, semua piksel-piksel yang merupakan bagian dari image pejalan kaki harus di*assign* ke dalam segmen "Pejalan Kaki" (terdapat satu segmen yang sudah dibentuk sebelummya berisi semua pejalan kaki). Sedangkan pada **Segmentasi *Instance***, semua piksel-piksel yang merupakan bagian dari objek individual yang sama akan di*assign* pada segmen yang sama. Pada kasus ini, akan terdapat segmen yang berbeda untuk setiap pejalan kaki. *State of The Art* untuk segmentasi sematik

dan *instance* saat ini banyak menggunakan arsitektur yang kompleks berdasarkan *Convolutional Neural Networks* (lihat modul selanjutnya). Di bagian ini kita akan melakukan sesuatu yang lebih sederhana, yaitu segementasi warna. Kita akan *assign* piksel-piksel pada segmen yang sama jika mempunyai warna yang serupa. Teknik semacam ini bisa dianggap sudah cukup untuk beberapa aplikasi. Sebagai contoh, jika kita akan melakukan analisa image-image dari satelit untuk mengukur sebarapa luas suatu hutan pada area tertentu, maka segmentasi warna akan cukup memenuhi untuk keperluan ini.

Pertama, gunakan fungsi `imread()` dari `Matplotlib` untuk *load* image (lihat gambar original sebelah kiri atas pada gambar di bawah hasil eksekusi program). Image akan direspresentasikan sebagai array 3 dimensi (3D). dimensi pertama adalah ukuran tinggi, kedua adalah ukuran lebar dan ketiga adalah jumlah *channel* warna, dalam hal ini RGB yaitu warna merah (*red*), hijau (*green*) dan biru (*blue*). Dengan kata lain untuk setiap piksel akan terdapat vektor 3D yang berisi intensitas warna merah, hijau dan biru, yang masing-masing bernilai antara 0,0-1,0 (atau antara 0 s/d 255, jika digunakan `imageio.imread()`). Beberapa image mempunyai *channel* warna yang lebih sedikit, misalkan image *grayscale* (satu *channel*). Dan ada beberapa jenis image juga mempunyai lebih banyak *channel* warna, seperti image-image dengan penambahan *channel* alpha untuk image satelit yang berisi *channel-channel* untuk banyak frekuensi cahaya (mis. infrared)

[105]:
```
import os
import urllib
# Where to save the figures
PROJECT_ROOT_DIR = "."
images_path = os.path.join(PROJECT_ROOT_DIR, "images", "unsupervised_learning")
os.makedirs(images_path, exist_ok=True)
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
filename = "ladybug.png"
print("Downloading", filename)
url = DOWNLOAD_ROOT + "images/unsupervised_learning/" + filename
urllib.request.urlretrieve(url, os.path.join(images_path, filename))
```

```
Downloading ladybug.png
```

[105]:
```
('./images/unsupervised_learning/ladybug.png',
 <http.client.HTTPMessage at 0x1a24a3ca90>)
```

[104]:
```
from matplotlib.image import imread
image = imread(os.path.join(images_path, filename))
image.shape
```

[104]:
```
(533, 800, 3)
```

Kode berikut mengubah bentuk (`reshape()`) array untuk mendapatkan list panjang dari warna RGB, kemudian melakukan *clustering* warna-warna menggunakan *K-Means*. Sebagai contoh proses yang terjadi pada kode di bawah, misalkan algoritma akan mengidentifikasi warna *cluster* untuk semua nuansa hijau. Untuk setiap warna (mis. hijau gelap) akan dicari warna rata-rata dari setiap warna *cluster* dari piksel. Dalam hal ini bisa jadi untuk semua warna dengan nuansa hijau akan direpresentasikan dengan satu warna hijau muda (hasil rata-rata). Akhirnya, dilakukan

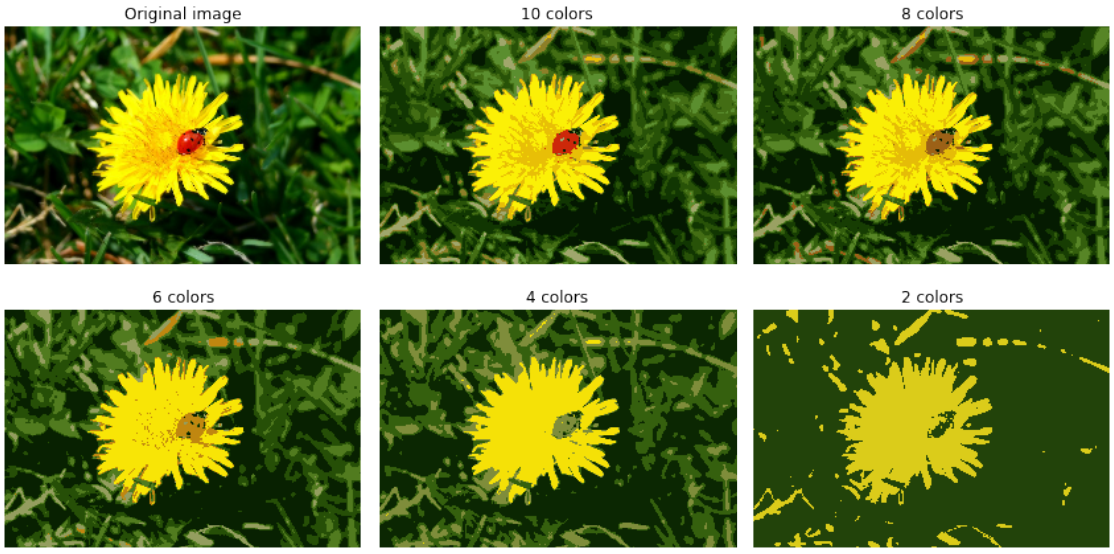
`reshape()` kembali list panjang dari warna tersebut untuk mengembalikan ke bentuk image original.

[110]:
```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8, random_state=42).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

Kode di bawah menggunakan jumlah *cluster* bervariasi dari 2 s/d 10. Hasil segmentasi gambar bisa dilihat di bawah.

[107]:
```
segmented_imgs = []
n_colors = (10, 8, 6, 4, 2)
for n_clusters in n_colors:
    kmeans = KMeans(n_clusters=n_clusters, random_state=42).fit(X)
    segmented_img = kmeans.cluster_centers_[kmeans.labels_]
    segmented_imgs.append(segmented_img.reshape(image.shape))
```

[131]:
```
plt.figure(figsize=(15,7.5))
plt.subplots_adjust(wspace=0.05, hspace=0.1)

plt.subplot(231)
plt.imshow(image)
plt.title("Original image")
plt.axis('off')

for idx, n_clusters in enumerate(n_colors):
    plt.subplot(232 + idx)
    plt.imshow(segmented_imgs[idx])
    plt.title("{} colors".format(n_clusters))
    plt.axis('off')
plt.show()
```



Kita dapat melakukan eksperimen dengan jumlah *cluster* yang berbeda-beda seperti yang ditunjukan pada gambar di atas. Terlihat dari gambar, jika kita menggunakan *cluster* lebih kecil dari 8 maka kumbang (*ladybug*) yang ada di gambar tersebut gagal teridentifikasi sebagai *cluster* warna merah, tetapi berbaur dengan warna sekitar. Hal ini karena *K-Means* cenderung memilih *cluster-cluster* dengan ukuran yang sama. Sedangkan *ladybug* berukuran kecil, jauh lebih kecil dibandingkan dengan bagian image yang lain, sehingga *K-Means* gagal untuk mendedikasikan sebuah *cluster* untuk *ladybug*.

3.2 Penggunaan *Clustering* untuk *Preprocessing*

Clustering bisa digunakan sebagai pendekatan yang efisien untuk pengurangan dimensi (*dimensionality reduction*, lihat *chapter* selanjutnya), atau sebagai langkah *preprocessing* sebelum algoritma *supervised learning* digunakan. Sebagai contoh penggunaan *clustering* untuk *preprocessing*, kita akan kembali melihat dataset dijit, yaitu dataset yang menyerupai dataset MNIST yang terdiri dari 1797 gambar *grayscale* dengan ukuran 8×8 yang merepresentasikan dijit 0 s/d 9. Pertama, load dataset tersebut.

[111]:
```
from sklearn.datasets import load_digits
```

[112]:
```
X_digits, y_digits = load_digits(return_X_y=True)
```

Split data ke dalam training set dan test set, seperti yang telah kita lakukan di chapter sebelumnya.

[113]:
```
from sklearn.model_selection import train_test_split
```

[114]:
```
X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits,␣
↪random_state=42)
```

Kemudian kita gunakan model regresi logistik.

[115]:
```
from sklearn.linear_model import LogisticRegression
```

[116]:
```
log_reg = LogisticRegression(multi_class="ovr", solver="lbfgs", max_iter=5000,␣
↪random_state=42)
log_reg.fit(X_train, y_train)
```

[116]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=5000,
                   multi_class='ovr', n_jobs=None, penalty='l2',
                   random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

Keakuratan dari klasifikasi akan diperoleh:

[118]:
```
log_reg.score(X_test, y_test)
```

[118]:
```
0.9688888888888889
```

Hasil di atas akan kita jadikan sebagai baseline yaitu 96,9%. Kita akan lihat bahwa dengan menggunakan *K-Means* sebagai *preprocessing* maka keakuratan akan meningkat. Kita gunakan `pipeline` yang akan melakukan *clustering* data terlebih dahulu ke dalam 50 *cluster* dan menggantikan image-image dengan jarak masing-masing terhadap 50 *cluster* ini. Selanjutnya model regresi logistik akan diterapkan.

[119]:
```
from sklearn.pipeline import Pipeline
```

[120]:
```
pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50, random_state=42)),
    ("log_reg", LogisticRegression(multi_class="ovr", solver="lbfgs",␣
 ↪max_iter=5000, random_state=42)),
])
pipeline.fit(X_train, y_train)
```

[120]:
```
Pipeline(memory=None,
         steps=[('kmeans',
                 KMeans(algorithm='auto', copy_x=True, init='k-means++',
                        max_iter=300, n_clusters=50, n_init=10, n_jobs=None,
                        precompute_distances='auto', random_state=42,
                        tol=0.0001, verbose=0)),
                ('log_reg',
                 LogisticRegression(C=1.0, class_weight=None, dual=False,
                                    fit_intercept=True, intercept_scaling=1,
                                    l1_ratio=None, max_iter=5000,
                                    multi_class='ovr', n_jobs=None,
                                    penalty='l2', random_state=42,
                                    solver='lbfgs', tol=0.0001, verbose=0,
                                    warm_start=False))],
         verbose=False)
```

[121]:
```
pipeline.score(X_test, y_test)
```

[121]:
```
0.98
```

Terlihat bahwa skor membaik menjadi 98%, sehingga kita mengurangi *error rate* sebesar:

[123]:
```
1 - (1 - 0.98) / (1 - 0.968888)
```

[123]:
```
0.3571612239650296
```

atau sebesar $\approx$ 36%. Perbaikan tersebut kita peroleh dengan memilih jumlah *cluster* k secara sembarang, yang tentunya kita bisa mendapatkan hasil yang lebih baik.

Karena *K-Means* di sini merupakan langkah *preprocessing* dari rangkaian `pipeline` untuk klasifikasi, maka menemukan harga k yang baik akan lebih sederhana dibandingkan dengan sebelum-

nya. Tidak perlu melakukan analisa *silhouette* atau meminimalkan *inertia*. Harga *k* terbaik merupakan harga yang menghasilkan kinerja klasifikasi pada saat melakukan *cross-validation* (lihat di *chapter* sebelumnya). Kita dapat gunakan `GirdSearchCV` untuk menemukan jumlah *cluster* yang optimal pada kasus ini.

[124]:
```
from sklearn.model_selection import GridSearchCV
```

[132]:
```
param_grid = dict(kmeans__n_clusters=range(2, 20))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 18 candidates, totalling 54 fits
[CV] kmeans__n_clusters=2 ...

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

[CV] ... kmeans__n_clusters=2, total=   0.2s
[CV] kmeans__n_clusters=2 ...
[CV] ... kmeans__n_clusters=2, total=   0.1s
[CV] kmeans__n_clusters=2 ...

[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.2s remaining:    0.0s

[CV] ... kmeans__n_clusters=2, total=   0.1s
[CV] kmeans__n_clusters=3 ...
[CV] ... kmeans__n_clusters=3, total=   0.2s
[CV] kmeans__n_clusters=3 ...
[CV] ... kmeans__n_clusters=3, total=   0.2s
[CV] kmeans__n_clusters=3 ...
[CV] ... kmeans__n_clusters=3, total=   0.2s
[CV] kmeans__n_clusters=4 ...
[CV] ... kmeans__n_clusters=4, total=   0.2s
[CV] kmeans__n_clusters=4 ...
[CV] ... kmeans__n_clusters=4, total=   0.2s
[CV] kmeans__n_clusters=4 ...
[CV] ... kmeans__n_clusters=4, total=   0.2s
[CV] kmeans__n_clusters=5 ...
[CV] ... kmeans__n_clusters=5, total=   0.2s
[CV] kmeans__n_clusters=5 ...
[CV] ... kmeans__n_clusters=5, total=   0.2s
[CV] kmeans__n_clusters=5 ...
[CV] ... kmeans__n_clusters=5, total=   0.2s
[CV] kmeans__n_clusters=6 ...
[CV] ... kmeans__n_clusters=6, total=   0.3s
[CV] kmeans__n_clusters=6 ...
[CV] ... kmeans__n_clusters=6, total=   0.3s
[CV] kmeans__n_clusters=6 ...
[CV] ... kmeans__n_clusters=6, total=   0.3s
[CV] kmeans__n_clusters=7 ...
[CV] ... kmeans__n_clusters=7, total=   0.5s
```

```
[CV] kmeans__n_clusters=7 ...
[CV] ... kmeans__n_clusters=7, total=   0.4s
[CV] kmeans__n_clusters=7 ...
[CV] ... kmeans__n_clusters=7, total=   0.3s
[CV] kmeans__n_clusters=8 ...
[CV] ... kmeans__n_clusters=8, total=   0.4s
[CV] kmeans__n_clusters=8 ...
[CV] ... kmeans__n_clusters=8, total=   0.4s
[CV] kmeans__n_clusters=8 ...
[CV] ... kmeans__n_clusters=8, total=   0.4s
[CV] kmeans__n_clusters=9 ...
[CV] ... kmeans__n_clusters=9, total=   0.5s
[CV] kmeans__n_clusters=9 ...
[CV] ... kmeans__n_clusters=9, total=   0.5s
[CV] kmeans__n_clusters=9 ...
[CV] ... kmeans__n_clusters=9, total=   0.5s
[CV] kmeans__n_clusters=10 ...
[CV] ... kmeans__n_clusters=10, total=   0.6s
[CV] kmeans__n_clusters=10 ...
[CV] ... kmeans__n_clusters=10, total=   0.7s
[CV] kmeans__n_clusters=10 ...
[CV] ... kmeans__n_clusters=10, total=   0.8s
[CV] kmeans__n_clusters=11 ...
[CV] ... kmeans__n_clusters=11, total=   0.8s
[CV] kmeans__n_clusters=11 ...
[CV] ... kmeans__n_clusters=11, total=   0.8s
[CV] kmeans__n_clusters=11 ...
[CV] ... kmeans__n_clusters=11, total=   0.7s
[CV] kmeans__n_clusters=12 ...
[CV] ... kmeans__n_clusters=12, total=   0.8s
[CV] kmeans__n_clusters=12 ...
[CV] ... kmeans__n_clusters=12, total=   1.1s
[CV] kmeans__n_clusters=12 ...
[CV] ... kmeans__n_clusters=12, total=   1.3s
[CV] kmeans__n_clusters=13 ...
[CV] ... kmeans__n_clusters=13, total=   1.1s
[CV] kmeans__n_clusters=13 ...
[CV] ... kmeans__n_clusters=13, total=   1.0s
[CV] kmeans__n_clusters=13 ...
[CV] ... kmeans__n_clusters=13, total=   1.0s
[CV] kmeans__n_clusters=14 ...
[CV] ... kmeans__n_clusters=14, total=   1.3s
[CV] kmeans__n_clusters=14 ...
[CV] ... kmeans__n_clusters=14, total=   1.6s
[CV] kmeans__n_clusters=14 ...
[CV] ... kmeans__n_clusters=14, total=   1.1s
[CV] kmeans__n_clusters=15 ...
[CV] ... kmeans__n_clusters=15, total=   1.4s
```

```
[CV] kmeans__n_clusters=15 ...
[CV] ... kmeans__n_clusters=15, total=   1.2s
[CV] kmeans__n_clusters=15 ...
[CV] ... kmeans__n_clusters=15, total=   1.3s
[CV] kmeans__n_clusters=16 ...
[CV] ... kmeans__n_clusters=16, total=   1.5s
[CV] kmeans__n_clusters=16 ...
[CV] ... kmeans__n_clusters=16, total=   1.4s
[CV] kmeans__n_clusters=16 ...
[CV] ... kmeans__n_clusters=16, total=   1.3s
[CV] kmeans__n_clusters=17 ...
[CV] ... kmeans__n_clusters=17, total=   1.6s
[CV] kmeans__n_clusters=17 ...
[CV] ... kmeans__n_clusters=17, total=   1.8s
[CV] kmeans__n_clusters=17 ...
[CV] ... kmeans__n_clusters=17, total=   1.5s
[CV] kmeans__n_clusters=18 ...
[CV] ... kmeans__n_clusters=18, total=   1.9s
[CV] kmeans__n_clusters=18 ...
[CV] ... kmeans__n_clusters=18, total=   2.0s
[CV] kmeans__n_clusters=18 ...
[CV] ... kmeans__n_clusters=18, total=   2.1s
[CV] kmeans__n_clusters=19 ...
[CV] ... kmeans__n_clusters=19, total=   1.8s
[CV] kmeans__n_clusters=19 ...
[CV] ... kmeans__n_clusters=19, total=   1.7s
[CV] kmeans__n_clusters=19 ...
[CV] ... kmeans__n_clusters=19, total=   2.0s

[Parallel(n_jobs=1)]: Done  54 out of  54 | elapsed:   46.1s finished
```

```
[132]: GridSearchCV(cv=3, error_score=nan,
                    estimator=Pipeline(memory=None,
                                       steps=[('kmeans',
                                               KMeans(algorithm='auto', copy_x=True,
                                                      init='k-means++', max_iter=300,
                                                      n_clusters=50, n_init=10,
                                                      n_jobs=None,
                                                      precompute_distances='auto',
                                                      random_state=42, tol=0.0001,
                                                      verbose=0)),
                                              ('log_reg',
                                               LogisticRegression(C=1.0,
                                                                  class_weight=None,
                                                                  dual=False,
                                                                  fit_intercept=True,
                                                                  intercept_scaling=1,
                                                                  l1_ratio=None,
```

```
                                                    max_iter=5000,
                                                    multi_class='ovr',
                                                    n_jobs=None,
                                                    penalty='l2',
                                                    random_state=42,
                                                    solver='lbfgs',
                                                    tol=0.0001,
                                                    verbose=0,
                                                    warm_start=False))],
                                    verbose=False),
                 iid='deprecated', n_jobs=None,
                 param_grid={'kmeans__n_clusters': range(2, 20)},
                 pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
                 scoring=None, verbose=2)
```

[133]:
```
grid_clf.best_params_
```

[133]:
```
{'kmeans__n_clusters': 19}
```

[134]:
```
grid_clf.score(X_test, y_test)
```

[134]:
```
0.9622222222222222
```

Jika kita buat harga k semakin besar misalkan s/d 99 (dengan cara merubah bagian range dari 20 menjadi 100 seperti berikut `param_grid = dict(kmeans__n_clusters=range(2, 100))`), maka kita akan temukan bahwa dengan $k = 99$ akan diperoleh perbaikan keakuratan cukup signifikan, menjadi 98, 22% pada test set. Atau bahkan kita bisa eksplorasi k menjadi lebih besar lagi, untuk mendapatkan keakuratan yang lebih besar.

3.3 Penggunaan *Clustering* untuk *Semi-Supervised Learning*

Penggunaan lain dari *clustering* adalah pada *semi-supervised learning*, ketika kita mempunyai banyak *instance-instance* yang tidak mempunyai label dan hanya sebagian kecil yang mempunyai label. Akan kita lakukan training dari model regresi logistik pada sampel dari 50 *instance* yang mempunyai label dari dataset digits.

[135]:
```
n_labeled = 50
```

[137]:
```
log_reg = LogisticRegression(multi_class="ovr", solver="lbfgs", random_state=42)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

[137]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='ovr', n_jobs=None, penalty='l2',
                   random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

Kinerja model di atas pada test set adalah:

[138]:
```
log_reg.score(X_test, y_test)
```

[138]:
```
0.8333333333333334
```

Keakuratan hanya 83,3%. Hal ini tidak mengejutkan dibandingkan dengan hasil sebelumnya, karena kita hanya menggunakan sebagian dari training set. Kita akan melihat bahwa dengan menggunakan *clustering* akan diperoleh hasil yang lebih baik. Terlebih dahulu akan dilakukan *clustering* training set ke dalam 50 *cluster*. Kemudian untuk setiap *cluster* kita akan temukan image-image yang terdekat ke centroid. Kita akan menamainya ***image representatif***.

[142]:
```
k = 50
```

[143]:
```
kmeans = KMeans(n_clusters=k, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Kita akan melakukan ploting terhadap image-image representatif tersebut dan dilabeli secara manual.

[144]:
```
plt.figure(figsize=(8, 2))
for index, X_representative_digit in enumerate(X_representative_digits):
    plt.subplot(k // 10, 10, index + 1)
    plt.imshow(X_representative_digit.reshape(8, 8), cmap="binary",␣
 ↪interpolation="bilinear")
    plt.axis('off')
plt.show()
```

[152]:
```
# Pelabelan secara manual
y_representative_digits = np.array([
    0, 1, 3, 2, 7, 6, 4, 6, 9, 5,
    0, 2, 9, 5, 2, 7, 8, 8, 8, 6,
    3, 1, 5, 4, 5, 4, 0, 3, 2, 6,
    1, 7, 7, 9, 1, 8, 6, 5, 4, 8,
    5, 3, 3, 6, 7, 9, 7, 8, 4, 9])
```

Sekarang kita mempunyai training set dengan 50 *instance* yang dilabeli, dan masing-masing adalah representatif image dari setiap *cluster* bukan lagi *instance-instance* yang acak. Maka akan kita lihat kinerja meningkat.

[153]:
```
log_reg = LogisticRegression(multi_class="ovr", solver="lbfgs", max_iter=5000,␣
↪random_state=42)
log_reg.fit(X_representative_digits, y_representative_digits)
log_reg.score(X_test, y_test)
```

[153]:
```
0.8933333333333333
```

Terdapat peningkatan dari 83,3% menjadi 89,3%, meskipun kita hanya melakukan training terhadap 50 *instance*. Lebih jauh, kita dapat mempropagasikan label-label pada *instance-instance* dari *cluster* yang sama, yang disebut dengan ***label propagation***.

[157]:
```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]
```

[158]:
```
log_reg = LogisticRegression(multi_class="ovr", solver="lbfgs", max_iter=5000,␣
↪random_state=42)
log_reg.fit(X_train, y_train_propagated)
```

[158]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=5000,
                   multi_class='ovr', n_jobs=None, penalty='l2',
                   random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

[159]:
```
log_reg.score(X_test, y_test)
```

[159]:
```
0.9111111111111111
```

Terlihat bahwa kita mendapatkan perbaikan akurasi menjadi 91,11%, dibandingkan sebelumnya 89,3%. Contoh di atas menunjukan perbaikan yang diperoleh ketika proses *clustering* digunakan untuk membantu pada saat training set hanya mempunyai sedikit *instance* yang dilabeli.

4 *Density-Based Spatial Clustering Algorithm with noise* (DBSCAN)

Algoritma ini mendefinisikan *cluster* sebagai daerah kontinyu dengan kepadatan tinggi. Berikut penjelasan bagaimana DBSCAN bekerja:

1. Untuk setiap *instance*, algoritma akan menghitung berapa *instance-instance* lain yang berlokasi pada daerah dengan jarak kecil maksimum sebesar ϵ dari lokasi *instance* yang dimaksud. Daerah ini disebut dengan *instance's* ϵ *neighborhood*.
2. Jika sebuah *instance* memiliki *instance-instance* tetangga minimal sebanyak `min_samples` pada ϵ-*neighborhood*-nya (termasuk *instance* itu sendiri), maka *instance* tersebut dipertim-

bangkan sebagai *instance* inti (*core instance*). Dengan kata lain, *instance-instance* inti berlokasi pada daerah padat.

3. Semua *instance-instance* disekitaran *instance* inti dapat dinyatakan bahwa *instance-instance* tersebut berada pada *cluster* yang sama. Definisi sekitaran juga bisa memasukan *instance* inti yang lain, sehingga urutan panjang dari *instance* inti membentuk sebuah *cluster*.
4. Sembarang *instance* yang bukan *instance* inti dan tidak mempunyai tetangga *instance* inti di sekitarnya dapat dikategorikan sebagai anomali.

Algoritma berikut dapat bekerja dengan baik jika *cluster-cluster* cukup padat dan masing-masing dapat dipisahkan satu sama lain oleh daerah-daerah dengan kepadatan rendah. *Class* `DBSCAN` pada `Scikit Learn` cukup mudah untuk digunakan.

Berikut contoh penggunaannya pada dataset `moons` yang sudah kita gunakan sebelumya

[85]:
```
from sklearn.datasets import make_moons
import numpy as np
```

[86]:
```
X, y = make_moons(n_samples=1000, noise=0.05, random_state=42)
```

[87]:
```
from sklearn.cluster import DBSCAN
```

[88]:
```
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

[88]:
```
DBSCAN(algorithm='auto', eps=0.05, leaf_size=30, metric='euclidean',
metric_params=None, min_samples=5, n_jobs=None, p=None)
```

- Label-label setiap *instance* sekarang dapat diakses dengan menggunakan variabel `labels_instance`. Sebagai contoh untuk 10 data pertama.

[89]:
```
dbscan.labels_[:10]
```

[89]:
```
array([ 0,  2, -1, -1,  1,  0,  0,  0,  2,  5])
```

- Jika kita perhatikan terdapat *instance* dengan indeks *cluster* negatif (-1), yang artinya *instance* tersebut dikategorikan sebagai anomali oleh algoritma. Indeks-indeks dari *instance* inti dapat diakses pada variabel `core_sample_indices_`, dan *instance-instance* inti sendiri dapat diakses melalui variabel `components_` sesuai dengan kode berikut.

[90]:
```
len(dbscan.core_sample_indices_)
```

[90]:
```
808
```

[91]:
```
dbscan.core_sample_indices_[:10] # hanya 10 data pertama
```

[91]:
```
array([ 0,  4,  5,  6,  7,  8, 10, 11, 12, 13])
```

[92]:
```
dbscan.components_[:3] # hanya 3 data pertama
```

[92]:
```
array([[-0.02137124,  0.40618608],
[-0.84192557,  0.53058695],
[ 0.58930337, -0.32137599]])
```

- Hasil *clustering* dapat ditampilkan dengan program berikut.

[93]:
```
np.unique(dbscan.labels_)
```

[93]:
```
array([-1,  0,  1,  2,  3,  4,  5,  6])
```

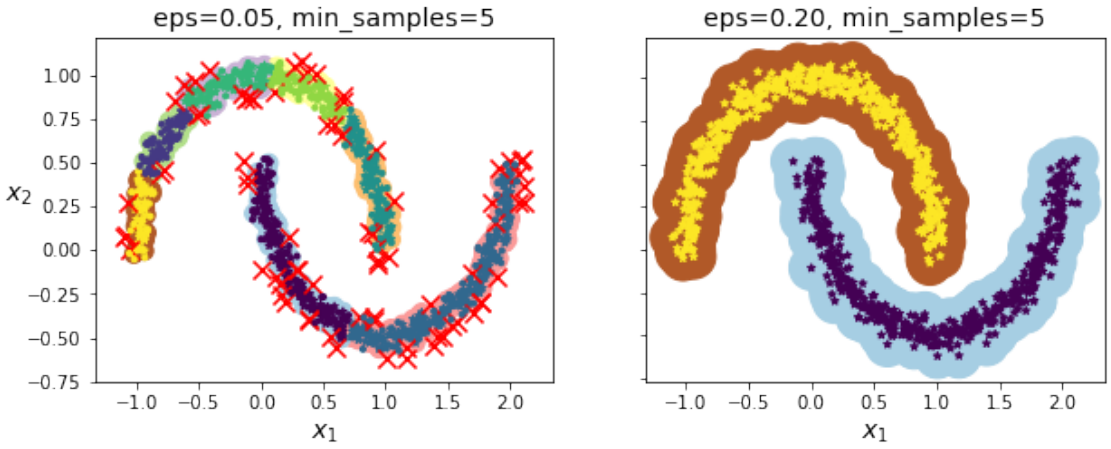
[94]:
```
dbscan2 = DBSCAN(eps=0.2)
dbscan2.fit(X)
```

[94]:
```
DBSCAN(algorithm='auto', eps=0.2, leaf_size=30, metric='euclidean',
metric_params=None, min_samples=5, n_jobs=None, p=None)
```

[95]:
```
def plot_dbscan(dbscan, X, size, show_xlabels=True, show_ylabels=True):
    core_mask = np.zeros_like(dbscan.labels_, dtype=bool)
    core_mask[dbscan.core_sample_indices_] = True
    anomalies_mask = dbscan.labels_ == -1
    non_core_mask = ~(core_mask | anomalies_mask)

    cores = dbscan.components_
    anomalies = X[anomalies_mask]
    non_cores = X[non_core_mask]

    plt.scatter(cores[:, 0], cores[:, 1],
    c=dbscan.labels_[core_mask], marker='o', s=size, cmap="Paired")
    plt.scatter(cores[:, 0], cores[:, 1], marker='*', s=20, c=dbscan.
↪labels_[core_mask])
    plt.scatter(anomalies[:, 0], anomalies[:, 1],
    c="r", marker="x", s=100)
    plt.scatter(non_cores[:, 0], non_cores[:, 1], c=dbscan.
↪labels_[non_core_mask], marker=".")
    if show_xlabels:
    plt.xlabel("$x_1$", fontsize=14)
    else:
    plt.tick_params(labelbottom=False)
    if show_ylabels:
    plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
    plt.tick_params(labelleft=False)
    plt.title("eps={:.2f}, min_samples={}".format(dbscan.eps, dbscan.
↪min_samples), fontsize=14)
```

[96]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 3.5))
```

```
plt.subplot(121)
plot_dbscan(dbscan, X, size=100)

plt.subplot(122)
plot_dbscan(dbscan2, X, size=600, show_ylabels=False)

plt.show()
```



Gambar sebelah kiri di atas menunjukan bahwa terdapat banyak anomali ditambah dengan 7 *cluster* yang berbeda, ketika harga $\epsilon = 0.05$. Tetapi kita dapat memperlebar jarak tetangga dari sebuah *instance* dengan cara merubah ϵ menjadi 0.2. Sehingga akan diperoleh gambar sebelah kanan. Selanjutnya kita akan memakai model dengan $\epsilon = 0.2$.

Class `DBSCAN` tidak mempunyai metode `predict()` meskipun mempunyai metode `fit_predict()`. Artinya `DBSCAN` tidak dapat digunakan untuk memprediksi/mengklasifikasikan *instance* baru termasuk pada *cluster* yang mana. Hal ini dikarenakan algoritma-algoritma klasifikasi yang berbeda mempunyai kinerja yang berlainan untuk kasus-kasus tertentu. Sehingga pengguna bisa memilih sendiri algoritma klasifikasi yang paling tepat setelah proses *clustering* menggunakan DBSCAN. Sebagai contoh, pada bagian ini kita akan menggunakan algoritma klasifikasi *K-nearest neighbor* dengan *class* `KNeighborClassifier` pada `Scikit Learn`.

[97]:
```
dbscan = dbscan2
```

[98]:
```
from sklearn.neighbors import KNeighborsClassifier
```

[99]:
```
knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```

[99]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
metric_params=None, n_jobs=None, n_neighbors=50, p=2,
weights='uniform')
```

- Sekarang jika dimisalkan terdapat beberapa *instance* baru, kita dapat memprediksi indeks *cluster* masing-masing dan bahkan bisa mengestimasi probabilitas keanggotan dari sebuah *cluster*.

[100]:
```
X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
knn.predict(X_new)
```

[100]:
```
array([1, 0, 1, 0])
```

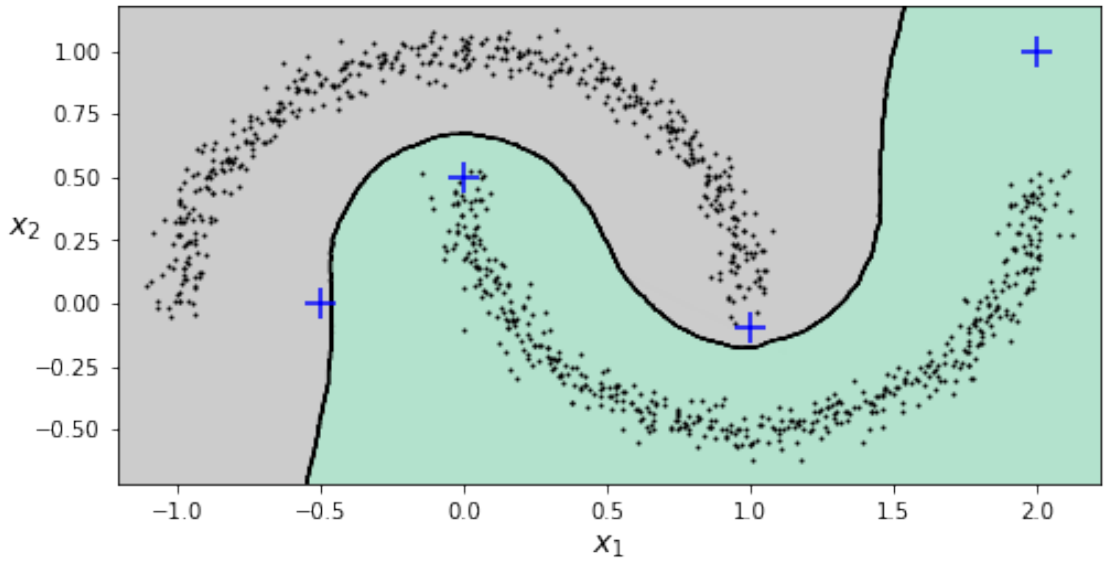
[101]:
```
knn.predict_proba(X_new)
```

[101]:
```
array([[0.18, 0.82],
[1.  , 0.  ],
[0.12, 0.88],
[1.  , 0.  ]])
```

Jika kita lihat pada langkah training di atas, kita hanya mempergunakan training set dari *instance-instance* inti, tetapi kita juga dapat melakukan training untuk keseluruhan *instance*, atau semua kecuali *instance-instance* anomali. Pilihan ini akan tergantung pada pekerjaan akhir yang ingin dilakukan.

- Batas keputusan untuk prediksi hasil training algoritma KNN dapat ditunjukan menggunakan program di bawah ini.

[103]:
```
plt.figure(figsize=(8, 4))
plot_decision_boundaries(knn, X, show_centroids=False)
plt.scatter(X_new[:, 0], X_new[:, 1], c="b", marker="+", s=200,␣
↪zorder=10)
plt.show()
```



Kita perhatikan gambar di atas, karena tidak terdapat anomali pada training set maka *classifier* selalu memilih sebuah *cluster*, meskipun *cluster* tersebut cukup jauh. Dengan cukup mudah kita bisa tentukan jarak maksimum sehingga *instance-instance* yang cukup jauh dapat dikategorikan sebagai anomali. Untuk melakukan ini dapat digunakan metode `keneighbors()` dari *class* `KNeighborsClassifier`. Jika diberikan satu set *instance* maka akan diperoleh jarak dan indeks dari *k nearest neighbors* pada training set (2 matriks, masing-masing dengan kolom sebanyak *k*).

[105]:
```
y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
y_pred[y_dist > 0.2] = -1
y_pred.ravel()
```

[105]:
```
array([-1,  0,  1, -1])
```

Secara singkat, DBSCAN merupakan algoritma yang sangat sederhana tetapi mempunyai kemampuan yang sangat baik (*powerful*) dalam mengidentifikasi jumlah *cluster* dengan bentuk bermacam-macam. DBSCAN sangat tahan terhadap data pencilan (*outliers*) dan hanya mempunyai dua *hyperparameter* saja, yaitu `eps` dan `min_samples`). Tetapi, jika kerapatan data bervariasi sangat signifikan pada *cluster-cluster*, maka tidak mungkin juga algoritma tersebut mengindikasikan dengan baik semua *cluster-cluster* yang ada.

5 *Gaussian Mixtures*

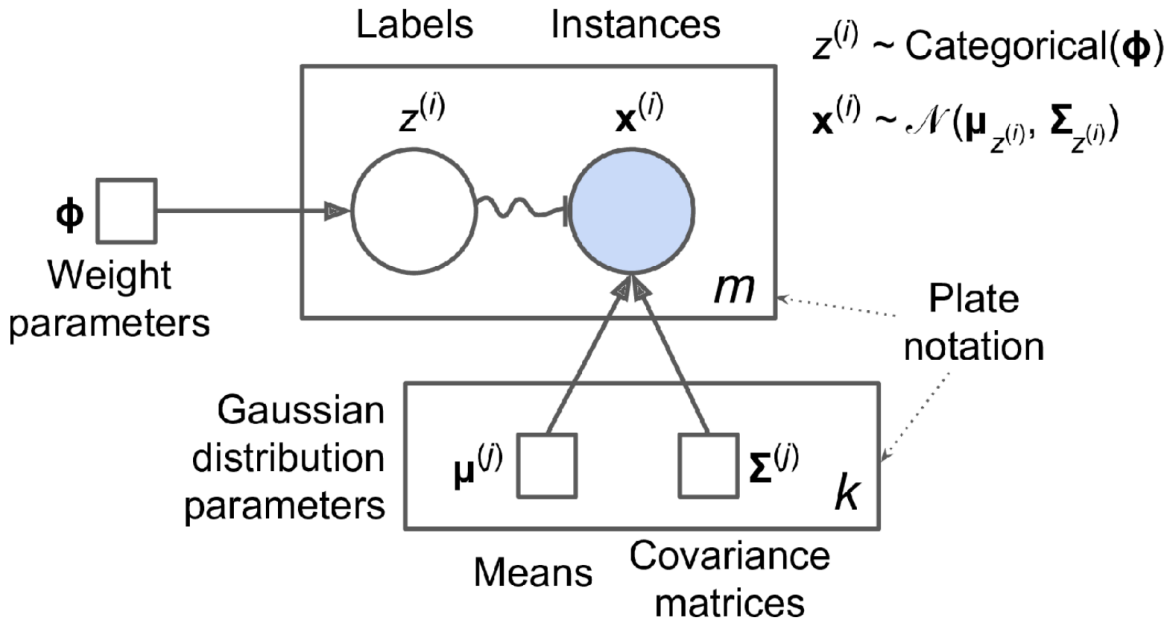
Model *Gaussian Mixture*, atau *Gaussian Mixture Model* (GMM) adalah model probabilistik yang mengasumsikan bahwa *instance* dibangkitkan dari gabungan (*mixture*) beberapa distribusi *Gaussian* dengan parameter yang tidak diketahui. Semua *instance-instance* yang dibangkitkan dari sebuah distribusi Gaussian membuat sebuah *cluster* yang berbentuk elips. Setiap *cluster* mempunyai bentuk elips, ukuran, kepadatan dan orientasi yang berbeda-beda. Ketika kita mengamati sebuah *instance*, kita akan tahu bahwa *instance* tersebut dibangkitkan dari salah satu distribusi Gaussian, tetapi kita tidak diberitahu yang mana, dan kita juga tidak tahu parameter-parameter dari distribusi tersebut.

Terdapat beberapa varian dari GMM. Yang paling sederhana dapat diimplementasikan pada *class* `GaussianMixture`, dimana kita harus ketahui terlebih dahulu jumlah distribusi Gaussiannya (k). Dataset X diasumsikan telah dibangkitkan melalui proses probabilistik berikut: * Untuk setiap *instance*, dipilihkan sebuah *cluster* secara acak diantara k *cluster* yang ada. Probabilitas memilih *cluster* ke-j didefinisikan sebagai pembobot *cluster*, $\phi^{(j)}$. Indeks *cluster* yang terpilih untuk instance ke-i dinotasikan dengan $z^{(i)}$. * Jika $z^{(i)} = j$, yang artinya *instance* ke-i telah di*assign* pada *cluster* ke-j, lokasi $x^{(i)}$ dari *instance* ini disampel secara acak dari distribusi Gaussian dengan mean $\mu^{(i)}$ dan matriks kovariansi $\Sigma^{(j)}$, yang biasa dinotasikan dengan $x^{(i)} \sim \mathcal{N}\left(\mu^{(j)}, \Sigma^{(j)}\right)$

Proses generatif ini bisa direpresentasikan dengan sebuah model grafis.

Interpretasi Gambar 6.8 adalah sebagai berikut:

- Bulatan merepresentasikan variabel acak, kotak merepresentasikan harga fix (mis. parameter-parameter dari model), dan kotak persegi besar disebut dengan *plates* yang mengindikasikan isinya diulang beberapa kali.



Gambar 6.8. Representasi grafis dari model Gaussian mixture, termasuk parameter-parameter (kotak), variabel acak (bulat) dan ketergantungan bersyaratnya atau conditional dependencies (panah tebal)

- Angka di sebelah kanan bawah pada masing-masing *plates* menyatakan berapakali isi dari *plates* diulang. Sehingga terdapat m variabel acak $z^{(i)}$ (dari $z^{(1)}$ ke $z^{(m)}$) dan m variabel acak $x^{(i)}$. Terdapat juga mean $\mu^{(j)}$ dan matriks kovariansi $\Sigma^{(j)}$ masing-masing sejumlah k. Selain itu, terakhir terdapat hanya satu vektor pembobot ϕ (berisi semua pembobot $\phi^{(1)}$ s/d $\phi^{(k)}$).
- Setiap variabel $z^{(i)}$ diambil dari distribusi kategorial dengan bobot ϕ. Setiap variabel $x^{(i)}$ diambil dari distribusi normal dengan mean dan matriks kovariansi yang terdifinisi oleh indeks *cluster*-nya $z^{(i)}$.
- Panah tebal merepresentasikan ketergantungan bersyarat. Sebagai contoh, distribusi probabilitas untuk setiap variabel acak $z^{(i)}$ tergantung pada vektor pembobot ϕ. Sebagai catatan, ketika panah melalui batasan *plate* (*plate boundary*) artinya itu berlaku untuk semua pengulangan pada *plate*. Misalkan, vektor ϕ berlaku untuk distibusi probabilitas semua variabel acak $x^{(1)}$ s/d $x^{(m)}$.
- Panah yang bergelombang dari $z^{(i)}$ ke $x^{(i)}$ merepresentasikan switch: Tergantung pada harga $z^{(i)}$, *instance* $x^{(i)}$ akan disampel dari distribusi Gaussian yang berbeda. Misalkan, jika $z^{(i)} = j$, maka $x^{(i)} \sim \mathcal{N}\left(\mu^{(j)}, \Sigma^{(j)}\right)$
- Node yang diarsir menyatakan bahwa harganya sudah diketahui. Sehingga, hanya variabel acak $x^{(i)}$ yang harganya diketahui dan disebut dengan *observed variables*. Sedangkan variabel random yang tidak diketahui harganya $z^{(i)}$ disebut dengan *latent variables*.

Pertanyaan selanjutnya adalah, apa yang bisa kita lakukan dengan model tersebut? Ketika kita mempunyai dataset X, kita dapat melakukan estimasi bobot ϕ dan semua parameter distribusi $\mu^{(1)}$ sampai $\mu^{(k)}$ dan $\Sigma^{(1)}$ sampai $\Sigma^{(k)}$. `Scikit Learn` dengan `GaussianMixture` membuat langkah-langkah ini mudah dilakukan, seperti contoh berikut.

- Membuat data sintetik

[9]:
```
from sklearn.datasets import make_blobs
X1, y1 = make_blobs(n_samples=1000, centers=((4, -4), (0, 0)),␣
↪random_state=42)
X1 = X1.dot(np.array([[0.374, 0.95], [0.732, 0.598]]))
X2, y2 = make_blobs(n_samples=250, centers=1, random_state=42)
X2 = X2 + [6, -8]
X = np.r_[X1, X2]
y = np.r_[y1, y2]
```

Let's train a Gaussian mixture model on the previous dataset:

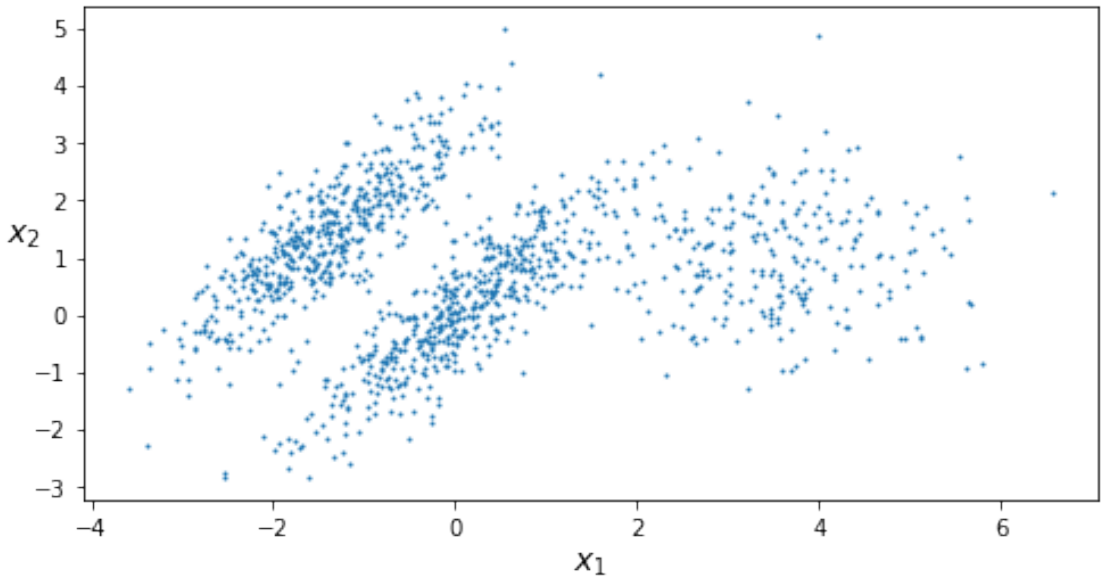
[10]:
```
from sklearn.mixture import GaussianMixture
```

[11]:
```
gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

[11]:
```
GaussianMixture(covariance_type='full', init_params='kmeans',␣
↪max_iter=100,
means_init=None, n_components=3, n_init=10,
precisions_init=None, random_state=42, reg_covar=1e-06,
tol=0.001, verbose=0, verbose_interval=10, warm_start=False,
weights_init=None)
```

[12]:
```
import matplotlib.pyplot as plt
def plot_clusters(X, y=None):
plt.scatter(X[:, 0], X[:, 1], c=y, s=1)
plt.xlabel("$x_1$", fontsize=14)
plt.ylabel("$x_2$", fontsize=14, rotation=0)
```

[13]:
```
plt.figure(figsize=(8, 4))
plot_clusters(X)
plt.show()
```



- Mendefinisikan model *Gaussian mixture* dan melakukan training pada dataset di atas.

[14]:
```
from sklearn.mixture import GaussianMixture
```

[15]:
```
gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

[15]:
```
GaussianMixture(covariance_type='full', init_params='kmeans',␣
↪max_iter=100,
means_init=None, n_components=3, n_init=10,
precisions_init=None, random_state=42, reg_covar=1e-06,
tol=0.001, verbose=0, verbose_interval=10, warm_start=False,
weights_init=None)
```

- Hasil estimasi parameter

[16]:
```
gm.weights_  # bobot
```

[16]:
```
array([0.39032584, 0.20961444, 0.40005972])
```

[17]:
```
gm.means_  # means
```

[17]:
```
array([[ 0.05145113,  0.07534576],
[ 3.39947665,  1.05931088],
[-1.40764129,  1.42712848]])
```

[18]:
```
gm.covariances_ # matriks kovariansi
```

[18]:
```
array([[[ 0.68825143,  0.79617956],
[ 0.79617956,  1.21242183]],

[[ 1.14740131, -0.03271106],
[-0.03271106,  0.95498333]],

[[ 0.63478217,  0.72970097],
[ 0.72970097,  1.16094925]]])
```

Menentukan apakah sudah konvergen dan berapakali iterasi dilakukan untuk estimasi parameter, dapat digunakan perintah berikut.

[19]:
```
gm.converged_
```

[19]:
```
True
```

[20]:
```
gm.n_iter_
```

[20]:
```
4
```

Terlihat bahwa algoritma telah berfungsi dengan baik. Bobot yang digunakan untuk membangkitkan data sintetik adalah $0.4, 0.4, 0.2$. Begitu juga dengan *means* dan matriks kovariansi sangat dekat dengan yang ditemukan algoritma.

Class dari `GaussianMixture` bergantung pada algoritma *Expectation Maximization* (EM-*algorithm*) yang memiliki banyak kesamaan dengan algoritma *K-Means* yang telah dijelaskan sebelumya. Cara kerja Algoritma EM secara singkat adalah sebagai berikut: 1. Menginisialisasi parameter *cluster* secara random 2. *Assign* setiap *instance* terhadap *cluster* (proses *expectation*) 3. Update kembali *cluster* (proses *maximization*). 4. Langkah 2 dan 3 diulang sampai konvergen.

Langkah di atas mirip dengan *K-Means*, perbedaannya algoritma EM bukan hanya mencari centroid dari *cluster-cluster* ($\mu^{(1)}$ sampai $\mu^{(k)}$), tetapi juga ukuran, bentuk dan orientasi ($\Sigma^{(1)}$ sampai $\Sigma^{(k)}$) dari masing-masing *cluster*, termasuk juga bobot ($\phi^{(1)}$ sampai $\phi^{(k)}$). Perbedaan lain dari EM adalah digunakannya *soft assignment* dari *instance* pada cluster, sedangkan *K-Means* menggunakan *hard assigment*. Pada *soft assigment*, untuk setiap *instance* saat langkah *expectation* dilakukan, algoritma akan melakukan estimasi probabilitas *instance* tersebut jika dimisalkan berasal dari masing-masing *cluster* yang ada. Kemudian pada langkah *maximization*, setiap parameter *cluster* diupadate menggunakan semua *instance* pada dataset, dimana setiap *instance* akan diboboti dengan probabilitas hasil estimasi setiap *instance* terhadap masing-masing *cluster*. Probabilitas-probabilitas setiap *instance* terhadap *cluster-cluster* ini disebut dengan *responsibilities* dari *cluster-cluster* terhadap *instance*. Pada langkah *maximization*, hasil update *cluster-cluster* akan sangat terpengaruh oleh probabilitas *instance* yang paling tinggi dari *cluster* tersebut.

Catatan. Algoritma EM mempunyai kelemahan yang sama dengan algoritma *K-Means* yaitu bisa konvergen tetapi pada solusi yang tidak optimal. Sehingga diperlukan eksekusi beberapa kali, kemudian dipilih hasil yang terbaik. Oleh sebab itu mengapa variabel `n_init =10` (perhatikan bahwa `n_init =1`) pada program di atas.

Setelah diperoleh estimasi lokasi, ukuran, bentuk, orientasi dan juga bobot relatif terhadap setiap *cluster*, model dapat melakukan *assignment* setiap *instance* terhadap sebuah *cluster* yang paling

mungkin (*hard clustering*) dengan menggunakan metode `predict()`, atau estimasi probabilitas bahwa *instance* tersebut berasal dari *cluster* tertentu (*soft clustering*) dengan `predict_proba()`.

[23]:
```
gm.predict(X)
```

[23]:
```
array([0, 0, 2, ..., 1, 1, 1])
```

[51]:
```
gm.predict_proba(X)
```

[51]:
```
array([[9.76815996e-01, 2.31833274e-02, 6.76282339e-07],
[9.82914418e-01, 1.64110061e-02, 6.74575575e-04],
[7.52377580e-05, 1.99781831e-06, 9.99922764e-01],
...,
[4.31902443e-07, 9.99999568e-01, 2.12540639e-26],
[5.20915318e-16, 1.00000000e+00, 1.45002917e-41],
[2.30971331e-15, 1.00000000e+00, 7.93266114e-41]])
```

Model *Gaussian mixture* merupakan model generatif (*generative model*), artinya kita dapat mengambil sampel *instance* baru dari model tersebut (sebagai catatan, mereka diurut berdasarkan indeks *cluster*).

[52]:
```
X_new, y_new = gm.sample(6)
X_new
```

[52]:
```
array([[-0.86951041, -0.32742378],
[ 0.29854504,  0.28307991],
[ 1.84860618,  2.07374016],
[ 3.98304484,  1.49869936],
[ 3.8163406 ,  0.53038367],
[-1.04030781,  0.78655831]])
```

[26]:
```
y_new
```

[26]:
```
array([0, 0, 1, 1, 1, 2])
```

Dimungkinkan juga untuk mengestimasi kerapatan dari model pada lokasi sembarang dengan menggunakan metode `score_samples()`. Dimana metode ini akan mengestimasi logaritmik dari *probability density function* (PDF) untuk setiap *instance* yang diberikan pada lokasi tersebut. Semakin besar skornya maka semakin besar kerapatannya.

[27]:
```
gm.score_samples(X)
```

[27]:
```
array([-2.60786904, -3.57094519, -3.3302143 , ..., -3.51359636,
-4.39793229, -3.80725953])
```

Jika kita hitung nilai eksponensial dari skor-skor ini, kita akan mendapatkan harga PDF pada lokasi *instance* yang diberikan. Nilai-nilai ini bukan probabilitas tapi rapat probabilitas (*probability density*), sehingga akan diperoleh nilai positif bukan hanya pada rentang 0 dan 1. Untuk menghitung probabilitas bahwa sebuah *instance* akan berada pada daerah tertentu, kita dapat mengin-

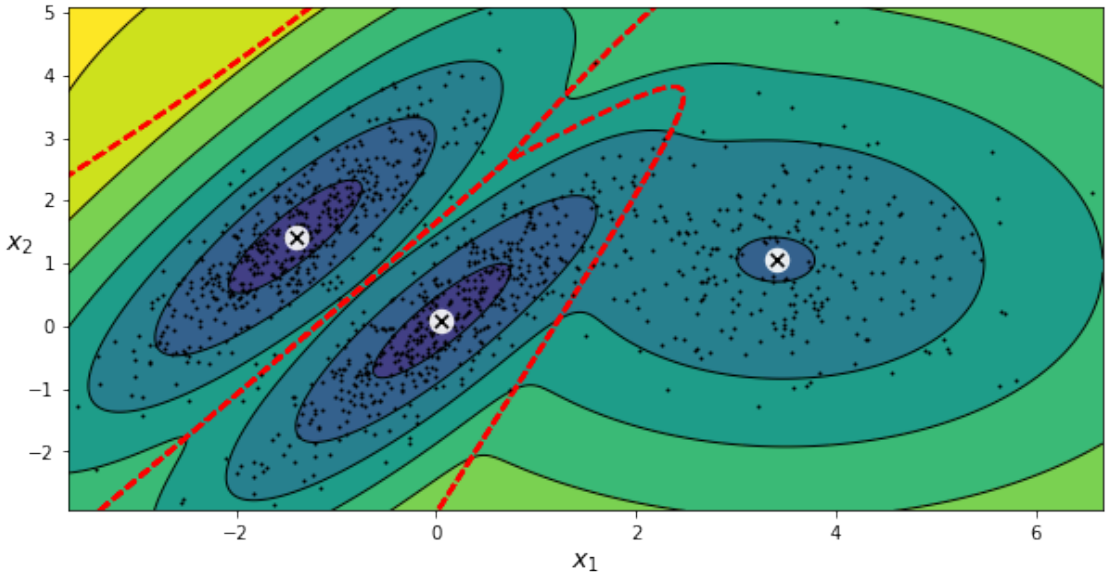
tegrasikan PDF pada daerah tersebut. Jika kita melakukan integrasi di keseluruhan ruang maka hasilnya harus sama dengan atau mendekati 1 (sesuai dengan konsep ruang sampel, probabilitas kejadian ruang sampel sama dengan 1).

Program berikut akan menunjukan *contour* kerapatan dari model ini, lokasi *means* dari tiap *cluster* dan batas keputusan (*decision boundary*).

[31]:
```
        from matplotlib.colors import LogNorm

        def plot_centroids(centroids, weights=None, circle_color='w',␣
↪cross_color='k'):
        if weights is not None:
        centroids = centroids[weights > weights.max() / 10]
        plt.scatter(centroids[:, 0], centroids[:, 1],
        marker='o', s=30, linewidths=8,
        color=circle_color, zorder=10, alpha=0.9)
        plt.scatter(centroids[:, 0], centroids[:, 1],
        marker='x', s=50, linewidths=50,
        color=cross_color, zorder=11, alpha=1)

        def plot_gaussian_mixture(clusterer, X, resolution=1000,␣
↪show_ylabels=True):
        mins = X.min(axis=0) - 0.1
        maxs = X.max(axis=0) + 0.1
        xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], resolution),
        np.linspace(mins[1], maxs[1], resolution))
        Z = -clusterer.score_samples(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)

        plt.contourf(xx, yy, Z,
        norm=LogNorm(vmin=1.0, vmax=30.0),
        levels=np.logspace(0, 2, 12))
        plt.contour(xx, yy, Z,
        norm=LogNorm(vmin=1.0, vmax=30.0),
        levels=np.logspace(0, 2, 12),
        linewidths=1, colors='k')

        Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        plt.contour(xx, yy, Z,
        linewidths=2, colors='r', linestyles='dashed')

        plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)
        plot_centroids(clusterer.means_, clusterer.weights_)

        plt.xlabel("$x_1$", fontsize=14)
        if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
```

```
        else:
        plt.tick_params(labelleft=False)
```

[33]:
```
        plt.figure(figsize=(10, 5))
        plot_gaussian_mixture(gm, X)
        plt.show()
```



Terlihat dari gambar di atas, algoritma telah mendapatkan solusi yang baik. Tentunya, kita telah melakukan penyederhanaan permasalahan karena data memang telah dibangkitkan dari distribusi Gaussian 2 dimensi (sayangnya data real biasanya tidak selalu Gaussian dan berdimensi kecil seperti yang telah kita bangkitkan). Dalam hal ini, kita juga sudah memberikan informasi awal yang benar menyangkut jumlah *cluster* k. Ketika dimensi besar, atau *cluster* berjumlah banyak, atau jumlah *instance* sedikit, algoritma EM akan mengalami kesulitan konvergen ke solusi yang optimal. Hal-hal ini bisa jadi mengharuskan kita untuk mengurangi kesulitan dengan membatasi jumlah parameter yang harus dipelajari oleh algoritma. Salah satu caranya adalah dengan membatasi range dari bentuk dan orientasi yang dapat dipunyai oleh sebuah *cluster*, dengan cara menerapkan batasan pada matriks kovariansi. Pada `Scikit Learn` bisa kita atur hyperparameter `covariance_type` pada salah satu nilai berikut:

- `covariance_type = "spherical"`: semua *cluster* harus berbentuk sferis, tetapi diperbolehkan mempunyai diameter yang berbeda-beda (berbeda variansi).
- `covariance_type = "diag"`: *Cluster-cluster* dapat berbentuk elips dengan ukuran sembarang, tetapi sumbu-sumbu elipsoid tersebut harus paralel pada sumbu koordinat (matriks kovariansi harus diagonal).
- `covariance_type = "tied"`: Semua *cluster* harus mempunyai bentuk, ukuran dan orientasi elips yang sama (semua *cluster* mempunyai matriks kovariansi yang sama).

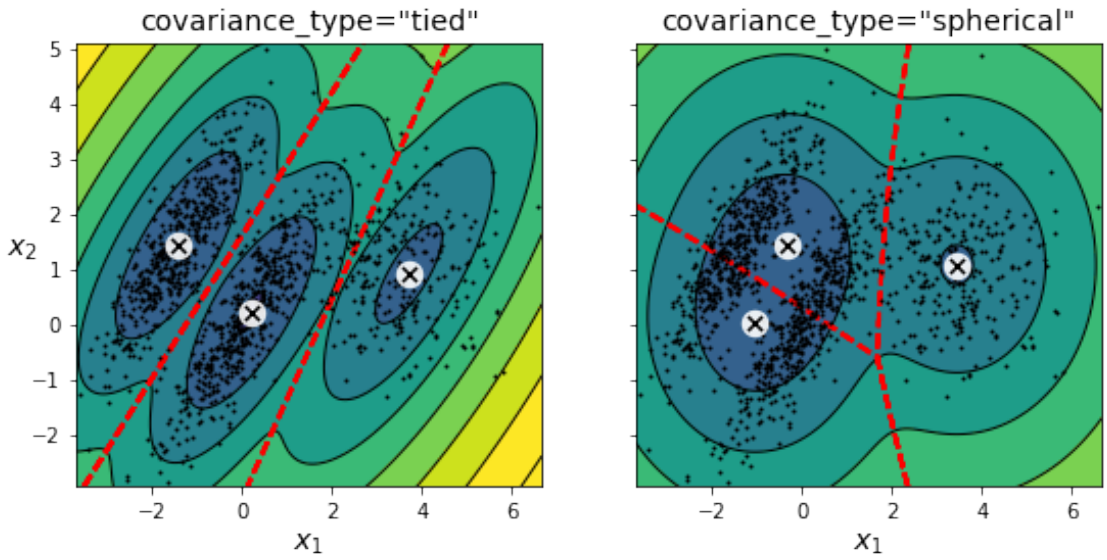
Secara *default* `covariance_type = "full"`, artinya setiap *cluster* dapat berbentuk, berukuran dan berorientasi bebas. Berikut hasil contoh hasil jika `covariance_type` berbeda dari default.

[42]:
```
        gm_full = GaussianMixture(n_components=3, n_init=10,␣
↪covariance_type="full", random_state=42)
        gm_tied = GaussianMixture(n_components=3, n_init=10,␣
↪covariance_type="tied", random_state=42)
        gm_spherical = GaussianMixture(n_components=3, n_init=10,␣
↪covariance_type="spherical", random_state=42)
        gm_diag = GaussianMixture(n_components=3, n_init=10,␣
↪covariance_type="diag", random_state=42)
        gm_full.fit(X)
        gm_tied.fit(X)
        gm_spherical.fit(X)
        gm_diag.fit(X)
```

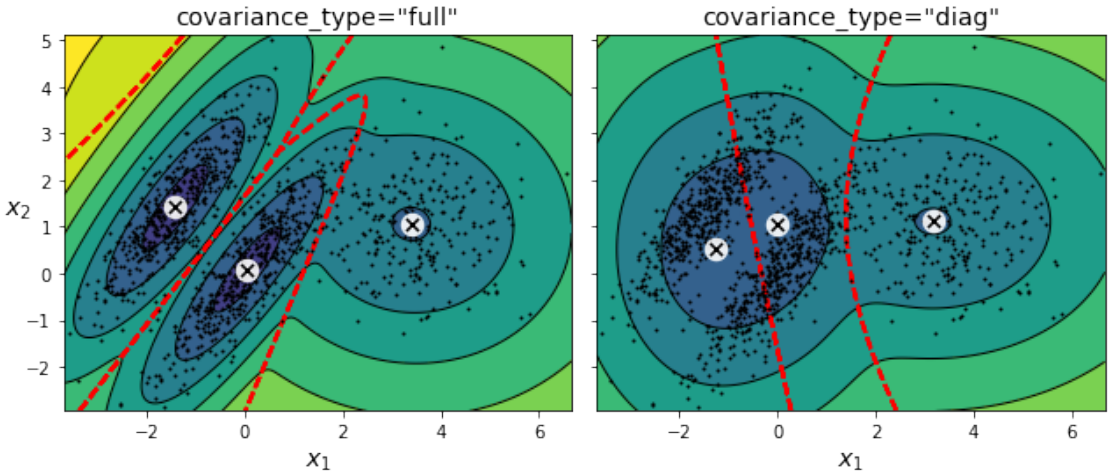
[42]:
```
        GaussianMixture(covariance_type='diag', init_params='kmeans',␣
↪max_iter=100,
        means_init=None, n_components=3, n_init=10,
        precisions_init=None, random_state=42, reg_covar=1e-06,
        tol=0.001, verbose=0, verbose_interval=10, warm_start=False,
        weights_init=None)
```

[43]:
```
        def compare_gaussian_mixtures(gm1, gm2, X):
        plt.figure(figsize=(9, 4))

        plt.subplot(121)
        plot_gaussian_mixture(gm1, X)
        plt.title('covariance_type="{}"'.format(gm1.covariance_type),␣
↪fontsize=14)

        plt.subplot(122)
        plot_gaussian_mixture(gm2, X, show_ylabels=False)
        plt.title('covariance_type="{}"'.format(gm2.covariance_type),␣
↪fontsize=14)
```

[45]:
```
        compare_gaussian_mixtures(gm_tied, gm_spherical, X)
        plt.show()
```



[46]:
```
compare_gaussian_mixtures(gm_full, gm_diag, X)
plt.tight_layout()
plt.show()
```



Catatan. Kompleksitas komputasi (*computational complexity*) dari training model `GaussianMixture` akan tergantung pada jumlah *instance* m, jumlah dimensi n, jumlah *cluster* k, dan batasan (*constraint*) pada matriks kovariansi. Jika `covariance_type` adalah `"spherical"` atau `"diag"`, maka kompleksitas komputasi yang dapat dinyatakan dengan notasi *Big-O* sebagai $O(kmn)$. Artinya kompleksitas komputasi berbading lurus dengan perkalian jumlah *instance*, jumlah dimensi dan jumlah *cluster*. Sedangkan jika `covariance_type` adalah `"tied"` atau `"full"` maka kompleksitas komputasi adalah $O(kmn^2 + kn^3)$ sehingga akan sulit melakukan penskalaan dengan jumlah *feature* n yang banyak.

5.1 Deteksi Anomali menggunakan *Gaussian Mixture*

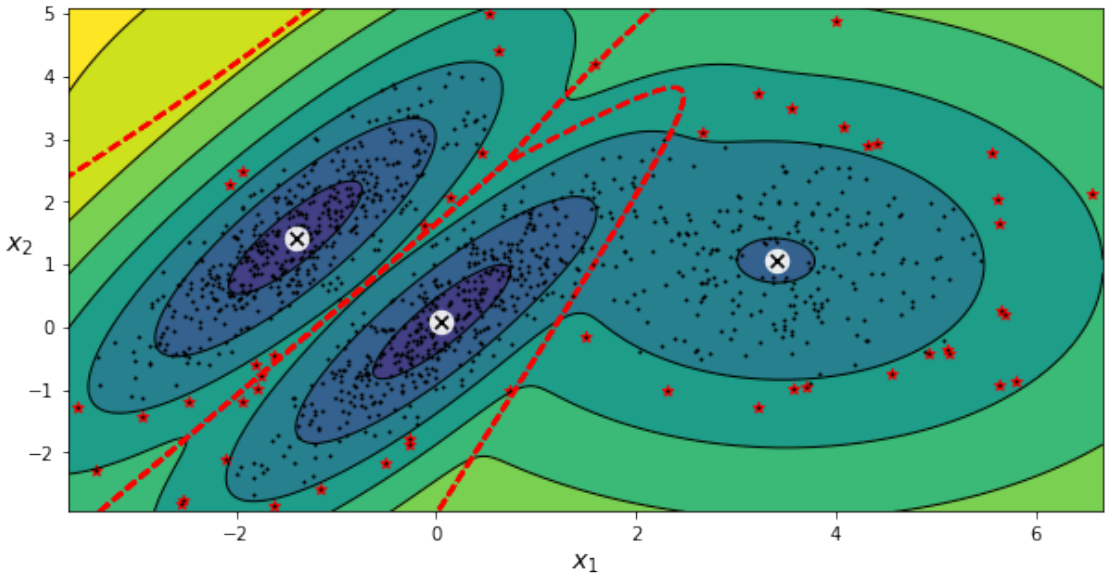
Deteksi anomali (*anomaly detection* atau *outlier detection*) adalah cara deteksi *instance* yang mempunyai deviasi yang cukup kuat dari norm mayoritas kebanyakan data. *Instance-instance* ini disebut dengan anomali atau pencilan (*outlier*), sementara *instance-instance* yang normal disebut dengan *inliers*. Deteksi anomali sangat berguna pada banyak aplikasi, seperti *fraud detection*, deteksi produk rusak di manufaktur, atau menghilangkan *outlier* dari dataset sebelum mentraining model lain (yang dapat memperbaiki kinerja cukup signifikan dari hasil model).

Penggunaan model *Gaussian mixture* untuk deteksi anomali cukup sederhana, yaitu menggunakan konsep *instance-instance* yang berlokasi pada daerah dengan kepadatan rendah maka dapat dikategorikan sebagai anomali. Kita harus definisikan ambang batas (*threshold*) dari kepadatan (*density threshold*) yang ingin digunakan. Misalkan, untuk perusahaan manufaktur yang berusaha untuk mendeteksi produk-produk yang rusak (*defective*), rasio produk rusak biasanya diketahui. Misalkan sama dengan 4%. Kemudian kita set *density threshold* dengan harga yang akan menghasilkan 4% *instance* berada di daerah dengan kepadatan dibawah *density threshold* tersebut. Jika ternyata kita mendapatkan terlalu banyak *false positive* (produk-produk baik yang dikatakan rusak), maka *threshold* dapat kita turunkan. Sebaliknya, jika terlalu banyak *false negative* (produk rusak dikatakan baik) maka *threshold* dapat ditambah. Hal ini berhubungan dengan *trade-off* antara *precision* dan *recall* yang sudah kita pelajari di Chapter 3.

Berikut adalah cara untuk mengidentifikasi *outlier* menggunakan percentile keempat dari kerapatan terendah sebagai *threshold* (yaitu, sekitar 4% dari *instance* akan dinyatakan sebagai anomali):

[47]:
```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

[50]:
```
plt.figure(figsize=(10, 5))

plot_gaussian_mixture(gm, X)
plt.scatter(anomalies[:, 0], anomalies[:, 1], color='r', marker='*')
plt.ylim(top=5.1)
plt.show()
```



Pada gambar di atas, data yang dikategorikan sebagai anomali menggunakan bintang berwarna merah.

Catatan. Model *Gaussian mixture* mencoba untuk fit semua data tidak terkecuali data *outlier*. Sehingga jika terlalu banyak data *outlier* maka akan membuat model menjadi bias terhadap definisi kenormalan (karena jumlah data normal menjadi sedikit). Pada kasus ini beberapa *outlier* akan dikategorikan salah sebagai normal. Jika ini terjadi, kita dapat mencoba fit model sekali, kemudian gunakan model tersebut untuk mendeteksi dan menghilangkan data *outlier* yang paling ekstrim. Dan fit kembali model menggunakan data yang telah dibersihkan tadi.

Seperti *K-Means*, algoritima `GaussianMixture` mengharuskan kita menspesifikasikan jumlah *cluster*, subsection berikut akan menjelaskan hal tersebut.

5.2 Memilih Jumlah *Cluster*

Pada *K-Means* kita bisa menggunakan *inertia* atau *silhoutte score* untuk memilih jumlah *cluster* yang tepat. Tetapi tidak mungkin menggunakan metrik ini pada *Gaussian Mixture* karena metrik tersebut tidak handal untuk *cluster-cluster* yang tidak sferis atau ukuran berbeda. Sehingga alternatifnya adalah menggunakan kriteria *Bayesian Information Criteria (BIC) atau Akaike Information Criteria (AIC), yang didefinisikan dengan Persamaan (6.2).

Persamaan (6.2) *Bayesian Information Criterion (BIC)* dan *Akaike Information Criterion (AIC)*

$$BIC = \log(m)p - 2\log\left(\hat{L}\right)$$

$$AIC = 2p - 2\log\left(\hat{L}\right)$$

Pada persamaan ini, m menyatakan jumlah *instance*, p adalah jumlah parameter yang dipelajari model, $\hat{L}$ adalah harga yang dimaksimalkan pada fungsi likelihood (*likelihood function*) dari model.

BIC dan AIC keduanya akan mempenalti model yang mempunyai banyak parameter-paremeter yang akan ditentukan (mis. karena lebih banyak *cluster*) dan memberikan *reward* pada model yang dapat *fitting* data dengan baik. Keduanya bisa jadi menghasilkan model yang sama. Ketika hasil berbeda, model yang dipilih oleh BIC cenderung lebih sederhana (lebih sedikit parameter) dibandingkan dengan model yang dipilih oleh AIC, tetap BIC tetap dapat *fitting* data cukup baik. Untuk menghitung BIC dan AIC, bisa menggunakan metode `bic()` dan `aic()`.

[53]:
```
gm.bic(X)
```

[53]:
```
8189.733705221635
```

[54]:
```
gm.aic(X)
```

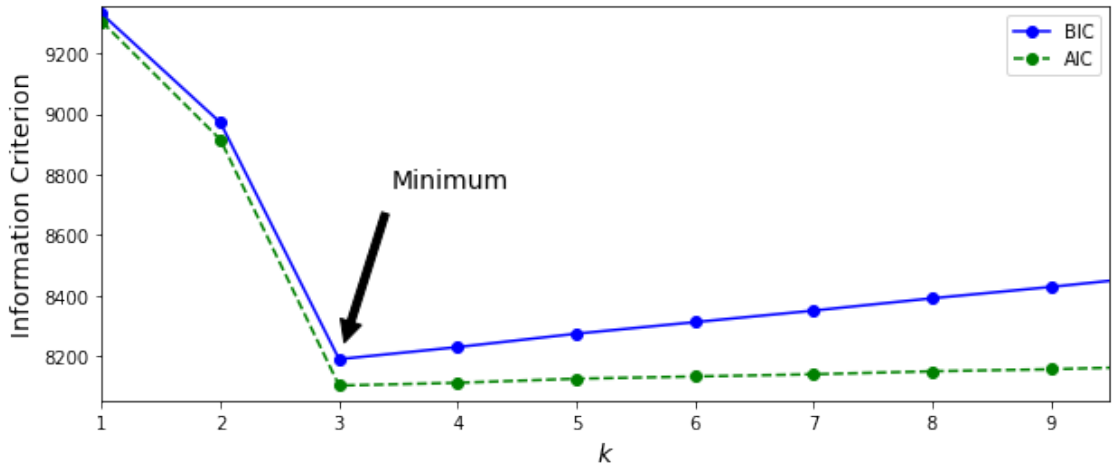
[54]:
```
8102.508425106597
```

Program berikut akan menunjukan BIC dan AIC sebagai fungsi dari jumlah *cluster*.

[55]:
```
gms_per_k = [GaussianMixture(n_components=k, n_init=10, random_state=42).
↪fit(X)
for k in range(1, 11)]
```

[56]:
```
bics = [model.bic(X) for model in gms_per_k]
aics = [model.aic(X) for model in gms_per_k]
```

[58]:
```
plt.figure(figsize=(10, 4))
plt.plot(range(1, 11), bics, "bo-", label="BIC")
plt.plot(range(1, 11), aics, "go--", label="AIC")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Information Criterion", fontsize=14)
plt.axis([1, 9.5, np.min(aics) - 50, np.max(aics) + 50])
plt.annotate('Minimum',
xy=(3, bics[2]),
xytext=(0.35, 0.6),
textcoords='figure fraction',
fontsize=14,
arrowprops=dict(facecolor='black', shrink=0.1)
)
plt.legend()
plt.show()
```



Dari gambar terlihat bahwa BIC dan AIC terendah diperoleh pada $k = 3$ yang boleh jadi merupakan pilihan terbaik pada kasus ini.

6 Algoritma-Algoritma Lain

`Scikit Learn` menyediakan beberapa algoritma *clustering* yang dapat kita gunakan. Pada modul ini tidak dapat dibahas satu persatu, tetapi akan dijelaskan secara singkat. Penjelasan detail dari masing-masing dapat dilihat di Algoritma-algoritma *clustering*.

- *Agglomerative Clustering*

  Cluster-cluster secara hierarki dibangun dari bawah ke atas. Pikirkan gelembung-gelembung kecil yang banyak mengambang pada air dan secara berangsur-angsur bergabung satu sama lain sehingga akan diperoleh sebuah grup yang besar dari gelembung-gelembung tadi. Serupa dengan itu, *Agglomerative Clustering* menghubungkan pasangan-pasangan *cluster* yang berdekatan (mulai dari *instance-instance* individu). Jika kita menggambar pohon dengan cabang untuk setiap pasangan *cluster* yang digabungkan (*merge*), kita akan peroleh *binary tree* dari *cluster-cluster* dimana daun-daunnya merepresentasikan *instance-instance* individu. Pendekatan ini dapat digunakan untuk kasus-kasus dengan jumlah *instance* atau *cluster* yang sangat besar. Algoritma ini dapat men-*capture cluster-cluster* dengan bentuk yang macam-macam, dan juga menghasilkan *cluster tree* yang fleksibel dan informatif.

- BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies)

  Algoritma BIRCH didesain khususnya untuk dataset yang sangat besar dan dapat bekerja lebih cepat dibandingkan dengan algoritma *batch K-Means* dengan hasil yang serupa selama jumlah *feature* tidak terlalu besar (< 20). Selama proses training, algoritma akan membangun struktur pohon yang berisi cukup informasi untuk meng*assign* sebuah *instance* baru ke dalam sebuah *cluster* tanpa harus menyimpan semua *instance* pada tree. Pendekatan ini hemat memory meskipun dapat bekerja dengan dataset yang sangat besar.

- *Mean-Shift*

Algoritma dimulai dengan menempatkan bulatan di tengah-tengah setiap *instance*. Untuk setiap bulatan akan dihitung mean berdasarkan *instance-instance* yang berlokasi di dalam bulatan tersebut, kemudian setelahnya *mean* tersebut dijadikan sebagai titik pusat dari lingkaran yang telah digeser (*shift*). Selanjutnya, mekanisme *mean-shifting* dilanjutkan sampai semua bulatan berhenti bergerak. Algoritma *Mean-Shift* menggeser bulatan-bulatan pada arah yang menghasilkan kepadatan lebih tinggi sampai dengan masing-masing menemukan kepadatan lokal yang maksimum. Akhirnya, semua *instance-instance* dengan bulatan-bulatan menuju pada tempat yang sama (atau cukup dekat) akan di*assign* pada *cluster* yang sama. *Computational complexity* dari *Mean-Shift* adalah $O(m^2)$, sehingga tidak cocok untuk dataset yang besar.

- *Affinity Propagation*

  Algoritma ini menggunakan sistem voting, dimana *instance-instance* melakukan vote untuk *instance-instance* yang serupa sebagai representatifnya. Setelah algoritma konvergen, setiap representatif dan *voter-voter*nya membentuk sebuah *cluster*. *Affinity Propagation* dapat mendeteksi jumlah *cluster* dengan ukuran yang berlainan. Sayangnnya, algoritma ini mempunyai kompleksitas yang tinggi yaitu $O(m^2)$, sehingga tidak cocok untuk dataset yang besar.

- *Spectral Clustering*

  ALgoritma ini membentuk matriks kesamaan (*similarity matrix*) antara *instance-instance* dan membuat versi *embedding* dengan dimensi lebih kecil (mengurangi ukuran dimensi). Kemudian algoritma ini mengimplementasikan algoritma *clustering* lain pada ruang dengan dimensi yang lebih kecil ini (`Scikit Learn` menggunakan *K-Means* pada langkah ini). *Spectral Clustering* dapat menangkap struktur *cluster* yang kompleks, dan dapat digunkan pula untuk memotong *graph* (mis. untuk mengidentifikasi *cluster-cluster* dari *friends* pada jaringan sosial). Algoritma ini tidak dapat diskalakan sehingga bekerja dengan baik pada dataset yang sangat besar dan tidak dapat berfungsi dengan baik jika *cluster-cluster* mempunyai ukuran yang berbeda-beda.

Pada *anomaly* atau *novelty detection* `Scikit Learn` juga menyediakan beberapa metode. Berikut penjelasan singkatnya, detail dapat dilihat pada Algoritma-algortima anomali.

- PCA (*Principle Component Analysis*)

  PCA dan teknik-teknik lain untuk *dimensionality reduction* (lihat Chapter selanjutnya) dengan menggunakan metode `inverse_transform()` dapat digunakan untuk deteksi anomali. Jika kita bandingkan error rekonstruksi dari *instance-instance* yang normal dan error rekonstruksi dari *instance outlier* maka error rekonstruksi dari *instance outlier* biasanya jauh lebih besar. Algoritma ini cukup sederhana dan cukup efisien untuk pendekatan deteksi anomali.

- *Fast-MCD (Minimum Covariance Determinant)*

  Diimpelementasikan dengan *class* `EllipticalEnvelope`, algoritma ini sangat berguna untuk deteksi *outlier*, terutama untuk membersihkan dataset. *Instance-instance* normal (*inliers*) diasumsikan berasal dari sebuah distribusi Gaussian (bukan *mixture*), selain itu diasumsikan juga bahwa dataset terkontaminasi oleh *outlier-outlier* yang tidak berasal dari distribusi Gaussian tersebut. Ketika algoritma melakukan estimasi parameter-parameter dari distribusi Gaussian, algoritma ini sangat berhati-hati untuk mengabaikan *instance* yang

mempunyai kemungkinan besar sebagai *outlier*. Teknik ini cukup baik untuk mengidentifikasi *outlier-outlier*.

- *Isolation Forest*

  Ini merupakan algoritma yang efisien untuk deteksi *outlier*, terutama untuk dataset dengan dimensi yang tinggi. Algoritma akan membangun *Random Forest* dimana setiap *Decision Tree* berkembang secara random. Pada setiap node, algoritma memilih *feature* secara random, kemudian memilih harga *threshold* secara random (antara harga *max* dan *min*) untuk membagi dataset menjadi dua. Dataset sedikit-sedikit akan terpotong-potong dengan cara ini, sampai dengan semua *instance-instance* terisolasi dari *instance-instance* lain. Anomali biasanya jauh dari *instance-instance* lain, sehingga secara rata-rata (di keseluruhan *decision tree*) akan terisolasi hanya dalam beberapa tahap dibandingkan dengan *instance-instance* normal.

- LOF (*Local Outlier Factor*)

  ALgoritma ini juga berkinerja baik untuk deteksi *outlier*, dengan cara membandingkan kerapatan dari *instance-instance* di sekeliling *instance* tertentu dengan kerapatan di sekitar tetangga-tetangganya. Dan anomali biasanya terisolasi lebih dibandingkan dengan sejumlah k tetangga terdekat.

- *One Class* SVM

  Algoritma ini cocok untuk *novelty detection*. Bekerja sangat baik terutama untuk dataset yang berdimensi tinggi. Tetapi seperti SVM biasanya tidak mampu diskalakan pada dataset yang besar.