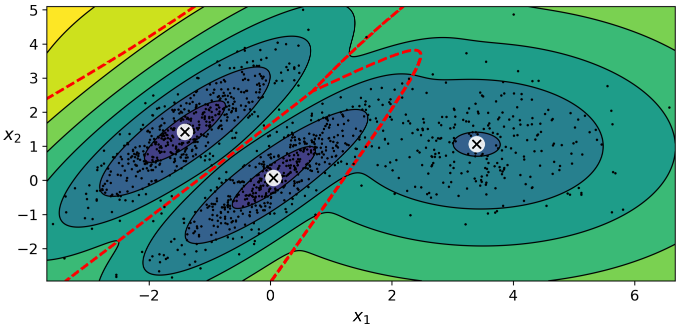

Figure 9-17. Cluster means, decision boundaries and density contours of a trained Gaussian mixture model

Nice! The algorithm clearly found an excellent solution. Of course, we made its task easy by actually generating the data using a set of 2D Gaussian distributions (unfortunately, real life data is not always so Gaussian and low-dimensional), and we also gave the algorithm the correct number of clusters. When there are many dimensions, or many clusters, or few instances, EM can struggle to converge to the optimal solution. You might need to reduce the difficulty of the task by limiting the number of parameters that the algorithm has to learn: one way to do this is to limit the range of shapes and orientations that the clusters can have. This can be achieved by imposing constraints on the covariance matrices. To do this, just set the `covariance_type` hyperparameter to one of the following values:

- `"spherical"`: all clusters must be spherical, but they can have different diameters (i.e., different variances).
- `"diag"`: clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the coordinate axes (i.e., the covariance matrices must be diagonal).
- `"tied"`: all clusters must have the same ellipsoidal shape, size and orientation (i.e., all clusters share the same covariance matrix).

By default, `covariance_type` is equal to `"full"`, which means that each cluster can take on any shape, size and orientation (it has its own unconstrained covariance matrix). Figure 9-18 plots the solutions found by the EM algorithm when `covariance_type` is set to `"tied"` or `"spherical`".

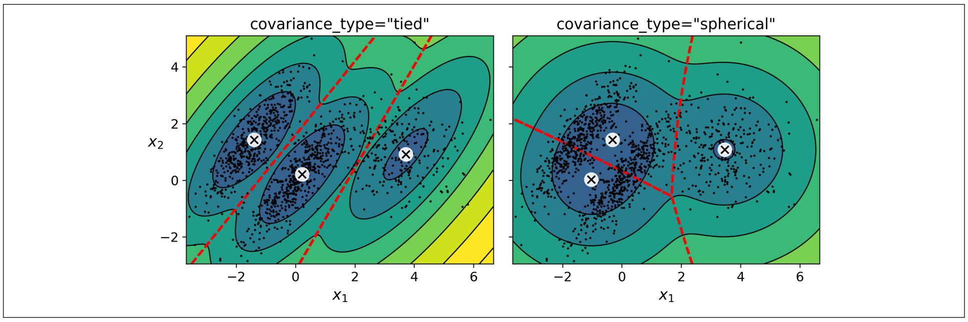

Figure 9-18. covariance_type_diagram

The computational complexity of training a `GaussianMixture` model depends on the number of instances *m*, the number of dimensions *n*, the number of clusters *k*, and the constraints on the covariance matrices. If `covariance_type` is `"spherical` or `"diag"`, it is O(*kmn*), assuming the data has a clustering structure. If `covariance_type` is `"tied"` or `"full"`, it is $O(kmn^2 + kn^3)$, so it will not scale to large numbers of features.

Gaussian mixture models can also be used for anomaly detection. Let's see how.

Anomaly Detection using Gaussian Mixtures

Anomaly detection (also called *outlier detection*) is the task of detecting instances that deviate strongly from the norm. These instances are of course called *anomalies* or *outliers*, while the normal instances are called *inliers*. Anomaly detection is very useful in a wide variety of applications, for example in fraud detection, or for detecting defective products in manufacturing, or to remove outliers from a dataset before training another model, which can significantly improve the performance of the resulting model.

Using a Gaussian mixture model for anomaly detection is quite simple: any instance located in a low-density region can be considered an anomaly. You must define what density threshold you want to use. For example, in a manufacturing company that tries to detect defective products, the ratio of defective products is usually well-known. Say it is equal to 4%, then you can set the density threshold to be the value that results in having 4% of the instances located in areas below that threshold density. If you notice that you get too many false positives (i.e., perfectly good products that are flagged as defective), you can lower the threshold. Conversely, if you have too many false negatives (i.e., defective products that the system does not flag as defective), you can increase the threshold. This is the usual precision/recall tradeoff (see Chapter 3). Here is how you would identify the outliers using the 4th percentile lowest density as the threshold (i.e., approximately 4% of the instances will be flagged as anomalies):

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

These anomalies are represented as stars on Figure 9-19:

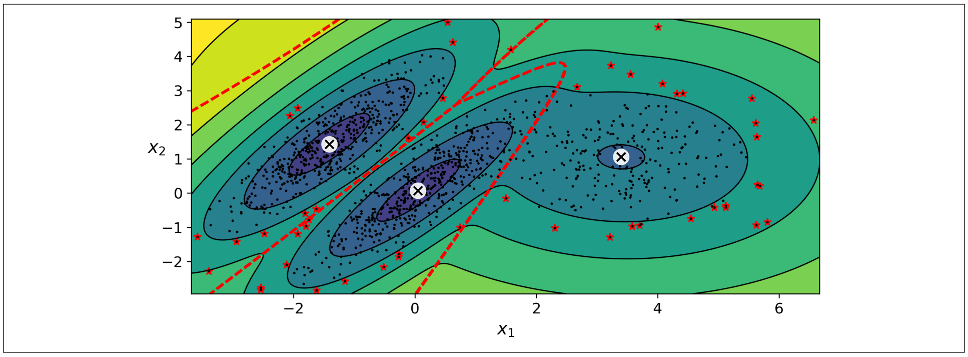

Figure 9-19. Anomaly detection using a Gaussian mixture model

A closely related task is *novelty detection*: it differs from anomaly detection in that the algorithm is assumed to be trained on a "clean" dataset, uncontaminated by outliers, whereas anomaly detection does not make this assumption. Indeed, outlier detection is often precisely used to clean up a dataset.

Gaussian mixture models try to fit all the data, including the outliers, so if you have too many of them, this will bias the model's view of "normality": some outliers may wrongly be considered as normal. If this happens, you can try to fit the model once, use it to detect and remove the most extreme outliers, then fit the model again on the cleaned up dataset. Another approach is to use robust covariance estimation methods (see the `EllipticEnvelope` class).

Just like K-Means, the `GaussianMixture` algorithm requires you to specify the number of clusters. So how can you find it?

Selecting the Number of Clusters

With K-Means, you could use the inertia or the silhouette score to select the appropriate number of clusters, but with Gaussian mixtures, it is not possible to use these metrics because they are not reliable when the clusters are not spherical or have different sizes. Instead, you can try to find the model that minimizes a *theoretical infor-*

mation criterion such as the *Bayesian information criterion* (BIC) or the *Akaike information criterion* (AIC), defined in Equation 9-1.

Equation 9-1. Bayesian information criterion (BIC) and Akaike information criterion (AIC)

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

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

- m is the number of instances, as always.
- p is the number of parameters learned by the model.
- $\hat{L}$ is the maximized value of the *likelihood function* of the model.

Both the BIC and the AIC penalize models that have more parameters to learn (e.g., more clusters), and reward models that fit the data well. They often end up selecting the same model, but when they differ, the model selected by the BIC tends to be simpler (fewer parameters) than the one selected by the AIC, but it does not fit the data quite as well (this is especially true for larger datasets).

Likelihood function

The terms "probability" and "likelihood" are often used interchangeably in the English language, but they have very different meanings in statistics: given a statistical model with some parameters $\boldsymbol{\theta}$, the word "probability" is used to describe how plausible a future outcome $\mathbf{x}$ is (knowing the parameter values $\boldsymbol{\theta}$), while the word "likelihood" is used to describe how plausible a particular set of parameter values $\boldsymbol{\theta}$ are, after the outcome $\mathbf{x}$ is known.

Consider a one-dimensional mixture model of two Gaussian distributions centered at -4 and +1. For simplicity, this toy model has a single parameter θ that controls the standard deviations of both distributions. The top left contour plot in Figure 9-20 shows the entire model $f(x; \theta)$ as a function of both x and θ. To estimate the probability distribution of a future outcome x, you need to set the model parameter θ. For example, if you set it to θ=1.3 (the horizontal line), you get the probability density function $f(x; \theta=1.3)$ shown in the lower left plot. Say you want to estimate the probability that x will fall between -2 and +2, you must calculate the integral of the PDF on this range (i.e., the surface of the shaded region). On the other hand, if you have observed a single instance x=2.5 (the vertical line in the upper left plot), you get the likelihood function noted $\mathscr{L}(\theta|x=2.5)=\text{f}(x=2.5; \theta)$ represented in the upper right plot.

In short, the PDF is a function of x (with θ fixed) while the likelihood function is a function of θ (with x fixed). It is important to understand that the likelihood function is *not* a probability distribution: if you integrate a probability distribution over all possible values of x, you always get 1, but if you integrate the likelihood function over all possible values of θ, the result can be any positive value.

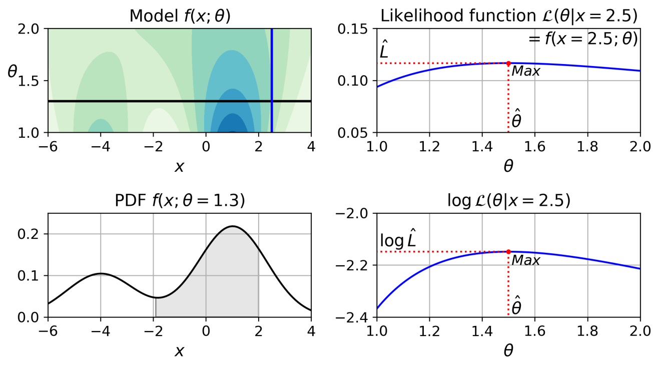

Figure 9-20. A model's parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right) and a log likelihood function (lower right)

Given a dataset **X**, a common task is to try to estimate the most likely values for the model parameters. To do this, you must find the values that maximize the likelihood function, given **X**. In this example, if you have observed a single instance x=2.5, the *maximum likelihood estimate* (MLE) of θ is $\hat{\theta}$=1.5. If a prior probability distribution g over θ exists, it is possible to take it into account by maximizing $\mathscr{L}(\theta|x)g(\theta)$ rather than just maximizing $\mathscr{L}(\theta|x)$. This is called maximum a-posteriori (MAP) estimation. Since MAP constrains the parameter values, you can think of it as a regularized version of MLE.

Notice that it is equivalent to maximize the likelihood function or to maximize its logarithm (represented in the lower right hand side of Figure 9-20): indeed, the logarithm is a strictly increasing function, so if θ maximizes the log likelihood, it also maximizes the likelihood. It turns out that it is generally easier to maximize the log likelihood. For example, if you observed several independent instances $x^{(1)}$ to $x^{(m)}$, you would need to find the value of θ that maximizes the product of the individual likelihood functions. But it is equivalent, and much simpler, to maximize the sum (not the product) of the log likelihood functions, thanks to the magic of the logarithm which converts products into sums: $\log(ab)=\log(a)+\log(b)$.

Once you have estimated $\hat{\theta}$, the value of θ that maximizes the likelihood function, then you are ready to compute $\hat{L} = \mathscr{L}\left(\hat{\theta}, \mathbf{X}\right)$. This is the value which is used to compute the AIC and BIC: you can think of it as a measure of how well the model fits the data.

To compute the BIC and AIC, just call the `bic()` or `aic()` methods:

```
>>> gm.bic(X)
8189.74345832983
>>> gm.aic(X)
8102.518178214792
```

Figure 9-21 shows the BIC for different numbers of clusters *k*. As you can see, both the BIC and the AIC are lowest when *k*=3, so it is most likely the best choice. Note that we could also search for the best value for the `covariance_type` hyperparameter. For example, if it is `"spherical"` rather than `"full"`, then the model has much fewer parameters to learn, but it does not fit the data as well.

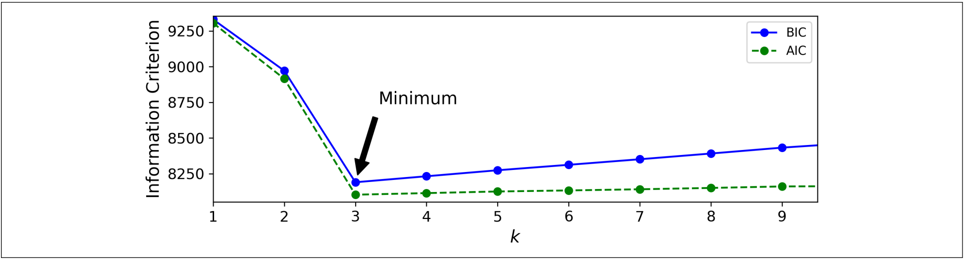

Figure 9-21. AIC and BIC for different numbers of clusters k

Bayesian Gaussian Mixture Models

Rather than manually searching for the optimal number of clusters, it is possible to use instead the `BayesianGaussianMixture` class which is capable of giving weights equal (or close) to zero to unnecessary clusters. Just set the number of clusters `n_components` to a value that you have good reason to believe is greater than the optimal number of clusters (this assumes some minimal knowledge about the problem at hand), and the algorithm will eliminate the unnecessary clusters automatically. For example, let's set the number of clusters to 10 and see what happens:

```
>>> from sklearn.mixture import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10, random_state=42)
>>> bgm.fit(X)
>>> np.round(bgm.weights_, 2)
array([0.4 , 0.21, 0.4 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

Perfect: the algorithm automatically detected that only 3 clusters are needed, and the resulting clusters are almost identical to the ones in Figure 9-17.

In this model, the cluster parameters (including the weights, means and covariance matrices) are not treated as fixed model parameters anymore, but as latent random variables, like the cluster assignments (see Figure 9-22). So **z** now includes both the cluster parameters and the cluster assignments.

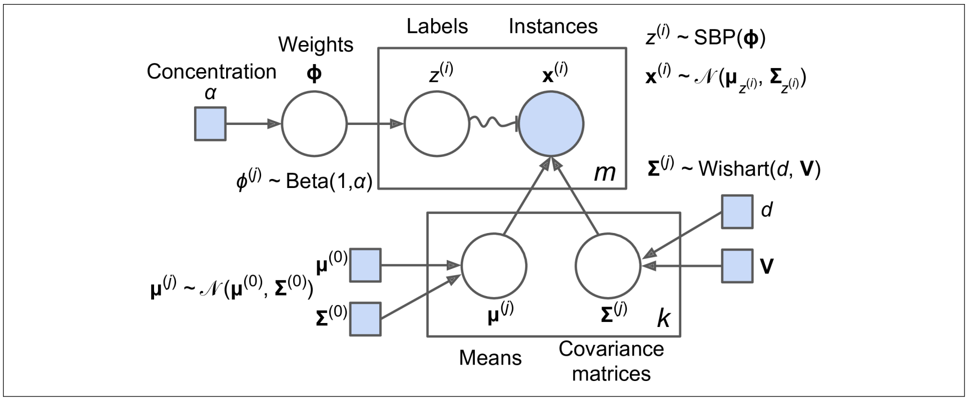
Labels
Instances
Weights
Concentration
α
$\boldsymbol{\phi}$
$z^{(i)}$
$\mathbf{x}^{(i)}$
$z^{(i)} \sim \text{SBP}(\boldsymbol{\phi})$
$\mathbf{x}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}_{z^{(i)}}, \boldsymbol{\Sigma}_{z^{(i)}})$
m
$\phi^{(j)} \sim \text{Beta}(1,\alpha)$
$\boldsymbol{\Sigma}^{(j)} \sim \text{Wishart}(d, \mathbf{V})$
d
$\mathbf{V}$
$\boldsymbol{\mu}^{(0)}$
$\boldsymbol{\mu}^{(j)} \sim \mathcal{N}(\boldsymbol{\mu}^{(0)}, \boldsymbol{\Sigma}^{(0)})$
$\boldsymbol{\Sigma}^{(0)}$
$\boldsymbol{\mu}^{(j)}$
$\boldsymbol{\Sigma}^{(j)}$
k
Means
Covariance matrices

Figure 9-22. Bayesian Gaussian mixture model

Prior knowledge about the latent variables **z** can be encoded in a probability distribution $p(\mathbf{z})$ called the *prior*. For example, we may have a prior belief that the clusters are likely to be few (low concentration), or conversely, that they are more likely to be plentiful (high concentration). This can be adjusted using the `weight_concentration_prior` hyperparameter. Setting it to 0.01 or 1000 gives very different clusterings (see Figure 9-23). However, the more data we have, the less the priors matter. In fact, to plot diagrams with such large differences, you must use very strong priors and little data.

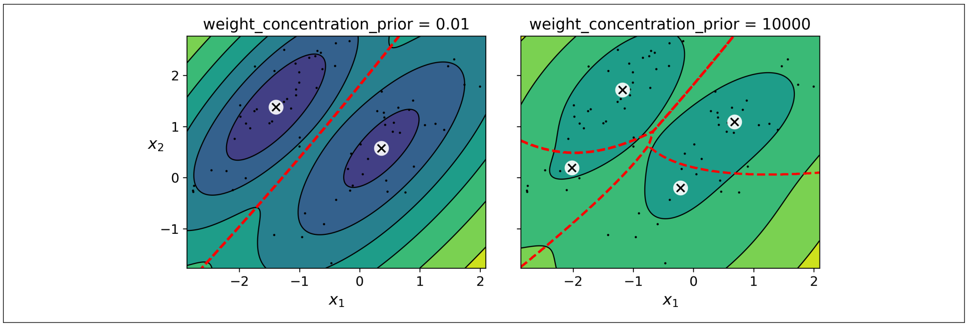

Figure 9-23. Using different concentration priors

The fact that you see only 3 regions in the right plot although there are 4 centroids is not a bug: the weight of the top-right cluster is much larger than the weight of the lower-right cluster, so the probability that any given point in this region belongs to the top-right cluster is greater than the probability that it belongs to the lower-right cluster, even near the lower-right cluster.

Bayes' theorem (Equation 9-2) tells us how to update the probability distribution over the latent variables after we observe some data $\mathbf{X}$. It computes the *posterior* distribution $p(\mathbf{z}|\mathbf{X})$, which is the conditional probability of $\mathbf{z}$ given $\mathbf{X}$.

Equation 9-2. Bayes' theorem

$$p(\mathbf{z}\,|\,\mathbf{X}) = \text{Posterior} = \frac{\text{Likelihood}\times\text{Prior}}{\text{Evidence}} = \frac{p(\mathbf{X}\,|\,\mathbf{z})\,p(\mathbf{z})}{p(\mathbf{X})}$$

Unfortunately, in a Gaussian mixture model (and many other problems), the denominator $p(\mathbf{x})$ is intractable, as it requires integrating over all the possible values of $\mathbf{z}$ (Equation 9-3). This means considering all possible combinations of cluster parameters and cluster assignments.

*Equation 9-3. The evidence p(**X**) is often intractable*

$$p\left(\mathbf{X}\right) = \int p(\mathbf{X}\,|\,\mathbf{z})p(\mathbf{z})d\mathbf{z}$$

This is one of the central problems in Bayesian statistics, and there are several approaches to solving it. One of them is *variational inference*, which picks a family of distributions $q(\mathbf{z};\,\boldsymbol{\lambda})$ with its own *variational parameters* $\boldsymbol{\lambda}$ (lambda), then it optimizes these parameters to make $q(\mathbf{z})$ a good approximation of $p(\mathbf{z}|\mathbf{X})$. This is achieved by finding the value of $\boldsymbol{\lambda}$ that minimizes the KL divergence from $q(\mathbf{z})$ to $p(\mathbf{z}|\mathbf{X})$, noted $\mathrm{D}_{\mathrm{KL}}(q\|p)$. The KL divergence equation is shown in (see Equation 9-4), and it can be rewritten as the log of the evidence ($\log\, p(\mathbf{X})$) minus the *evidence lower bound* (ELBO). Since the log of the evidence does not depend on q, it is a constant term, so minimizing the KL divergence just requires maximizing the ELBO.

*Equation 9-4. KL divergence from q(z) to p(z|**X**)*

$$\begin{aligned}
D_{KL}(q \parallel p) &= \mathbb{E}_q\left[\log \frac{q(\mathbf{z})}{p(\mathbf{z}\,|\,\mathbf{X})}\right] \\
&= \mathbb{E}_q[\,\log q(\mathbf{z}) - \log p(\mathbf{z}\,|\,\mathbf{X})] \\
&= \mathbb{E}_q\left[\log q(\mathbf{z}) - \log \frac{p(\mathbf{z},\mathbf{X})}{p(\mathbf{X})}\right] \\
&= \mathbb{E}_q[\,\log q(\mathbf{z}) - \log p(\mathbf{z},\mathbf{X}) + \log p(\mathbf{X})] \\
&= \mathbb{E}_q[\,\log q(\mathbf{z})] - \mathbb{E}_q[\,\log p(\mathbf{z},\mathbf{X})] + \mathbb{E}_q[\,\log p(\mathbf{X})] \\
&= \mathbb{E}_q[\,\log\ p(\mathbf{X})] - \left(\mathbb{E}_q[\,\log\ p(\mathbf{z},\mathbf{X})] - \mathbb{E}_q[\,\log\ q(\mathbf{z})]\right) \\
&= \log\ p(\mathbf{X}) - \text{ELBO} \\
&\quad \text{where ELBO} = \mathbb{E}_q[\,\log\ p(\mathbf{z},\mathbf{X})] - \mathbb{E}_q[\,\log\ q(\mathbf{z})]
\end{aligned}$$

In practice, there are different techniques to maximize the ELBO. In *mean field variational inference*, it is necessary to pick the family of distributions $q(\mathbf{z}; \boldsymbol{\lambda})$ and the prior $p(z)$ very carefully to ensure that the equation for the ELBO simplifies to a form that can actually be computed. Unfortunately, there is no general way to do this, it depends on the task and requires some mathematical skills. For example, the distributions and lower bound equations used in Scikit-Learn's `BayesianGaussianMixture` class are presented in the documentation. From these equations it is possible to derive update equations for the cluster parameters and assignment variables: these are then used very much like in the Expectation-Maximization algorithm. In fact, the computational complexity of the `BayesianGaussianMixture` class is similar to that of the `GaussianMixture` class (but generally significantly slower). A simpler approach to maximizing the ELBO is called *black box stochastic variational inference* (BBSVI): at each iteration, a few samples are drawn from q and they are used to estimate the gradients of the ELBO with regards to the variational parameters $\boldsymbol{\lambda}$, which are then used in a gradient ascent step. This approach makes it possible to use Bayesian inference with any kind of model (provided it is differentiable), even deep neural networks: this is called Bayesian deep learning.

If you want to dive deeper into Bayesian statistics, check out the *Bayesian Data Analysis* book by Andrew Gelman, John Carlin, Hal Stern, David Dunson, Aki Vehtari, and Donald Rubin.

Gaussian mixture models work great on clusters with ellipsoidal shapes, but if you try to fit a dataset with different shapes, you may have bad surprises. For example, let's see what happens if we use a Bayesian Gaussian mixture model to cluster the moons dataset (see Figure 9-24):

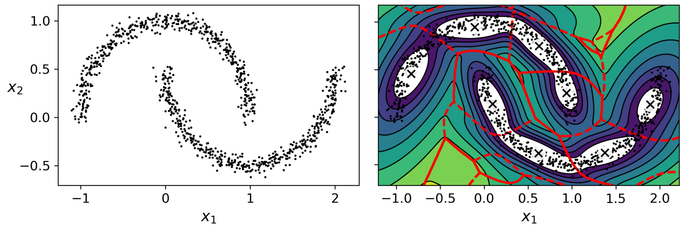

Figure 9-24. moons_vs_bgm_diagram

Oops, the algorithm desperately searched for ellipsoids, so it found 8 different clusters instead of 2. The density estimation is not too bad, so this model could perhaps be used for anomaly detection, but it failed to identify the two moons. Let's now look at a few clustering algorithms capable of dealing with arbitrarily shaped clusters.

Other Anomaly Detection and Novelty Detection Algorithms

Scikit-Learn also implements a few algorithms dedicated to anomaly detection or novelty detection:

- *Fast-MCD* (minimum covariance determinant), implemented by the `EllipticEnvelope` class: this algorithm is useful for outlier detection, in particular to cleanup a dataset. It assumes that the normal instances (inliers) are generated from a single Gaussian distribution (not a mixture), but it also assumes that the dataset is contaminated with outliers that were not generated from this Gaussian distribution. When it estimates the parameters of the Gaussian distribution (i.e., the shape of the elliptic envelope around the inliers), it is careful to ignore the instances that are most likely outliers. This gives a better estimation of the elliptic envelope, and thus makes it better at identifying the outliers.
- *Isolation forest*: this is an efficient algorithm for outlier detection, especially in high-dimensional datasets. The algorithm builds a Random Forest in which each Decision Tree is grown randomly: at each node, it picks a feature randomly, then it picks a random threshold value (between the min and max value) to split the dataset in two. The dataset gradually gets chopped into pieces this way, until all instances end up isolated from the other instances. An anomaly is usually far from other instances, so on average (across all the Decision Trees) it tends to get isolated in less steps than normal instances.
- *Local outlier factor* (LOF): this algorithm is also good for outlier detection. It compares the density of instances around a given instance to the density around its neighbors. An anomaly is often more isolated than its *k* nearest neighbors.
- *One-class SVM*: this algorithm is better suited for novelty detection. Recall that a kernelized SVM classifier separates two classes by first (implicitly) mapping all the instances to a high-dimensional space, then separating the two classes using a linear SVM classifier within this high-dimensional space (see Chapter 5). Since we just have one class of instances, the one-class SVM algorithm instead tries to separate the instances in high-dimensional space from the origin. In the original space, this will correspond to finding a small region that encompasses all the instances. If a new instance does not fall within this region, it is an anomaly. There are a few hyperparameters to tweak: the usual ones for a kernelized SVM, plus a margin hyperparameter that corresponds to the probability of a new instance being mistakenly considered as novel, when it is in fact normal. It works great, especially with high-dimensional datasets, but just like all SVMs, it does not scale to large datasets.