Lecture Book
B.Sc Data Science II

D. T. McGuiness, PhD

Version: 2024.1

Topics on Artificial Intelligence & Machine Learning

Current version is 2024.1.

This document includes the contents of Data Science II taught at MCI. This document is the part of Data Science & Machine Learning.

All relevant code of the document is done using *SageMath* v10.3 and Python v3.12.5.

This document was compiled with LuaTEX and all editing were done using
GNU Emacs using AUCTEX and org-mode package.

This document is based on the books on the topics of Machine Learning Data Science and Python Programming which includes *AI and Machine Learning for Coders* by L. Moroney , *Neural Networks and Deep Learning* by S. Aggarwal, *Python Machine Learning* by Raschka, et. al., *Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow* by A. Geron, *Machine Learning with Python Cookbook* by C. Albon, *CS229 Lecture Notes* by A. Ng, et. al., and *Lecture Notes on Machine Learning* by A. Migel et. al.,

The current maintainer of this work along with the primary lecturer
is D. T. McGuiness, PhD (dtm@mci4me.at).

Contents

Chapter 1

Support Vector Machines

Table of Contents

1.1 Introduction

A SVM is a powerful Machine Learning (ML) model, capable of performing **linear or nonlinear classification**, regression.

> It is even possible to implement **novelty detection** using SVM

SVMs most suitable for small to medium sized non-linear datasets (i.e., hundreds to thousands of instances), especially for classification tasks.

However, they don't scale very well to very large datasets as there are better classification models for them. Therefore SVM are mostly useful in scenarios with small to medium datasets.

1.2 Linear SVM Classification

As with most engineering and abstract concepts, the idea behind SVM is best explained with some visuals. Figure 1.1 shows part of the iris dataset that was introduced previously. The two (2) classes can easily and clearly separated with a straight line.

> This means the data is **linearly separable**.

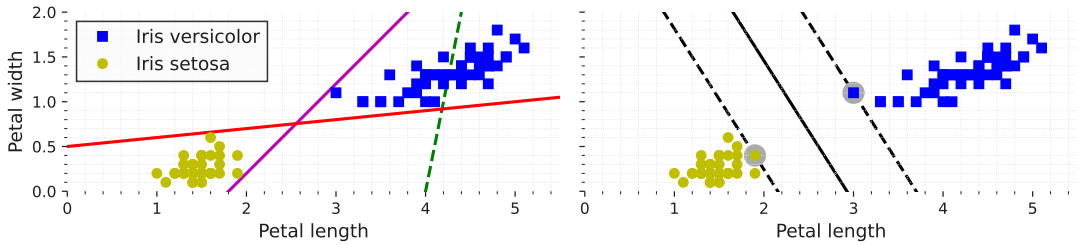

Figure 1.1: Large Margin Classifier

The left plot shows the decision boundaries of three possible linear classifiers. The model whose decision boundary is represented by the dashed line (- -) is so bad it does not even separate the classes properly.

The other two models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably not perform as well on new instances.

In contrast, the solid line in the plot on the right represents the **decision boundary** of an **SVM classifier**. This line not only separates the two classes but also stays as far away from the closest training instances as possible. Think of an SVM classifier as fitting the widest possible street, which in the plot is represented by the parallel dashed lines, between the classes.

This is called **large margin classification**.

Margin Classifier

A classifier which is able to give an associated distance from the decision boundary for each example. For instance, if a linear classifier is used, the distance of an example from the separating hyperplane is the margin of that example.

Notice that adding more training instances will not affect the decision boundary at all as the boundaries are fully determined (or *supported*) by the instances located on the edge of the boundaries.

These instances are called the **support vectors**.

SVMs are **sensitive to the feature scales**, as you can see in Figure 1.2. In the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (e.g., using `sklearn`'s `StandardScaler`), the decision boundary in the right plot looks much better.

1.2.1 Soft Margin Classification

If we impose all instances must be off the street and on the correct side, this is called **hard margin classification**. There are two (2) main issues with hard margin classification:

- It only works if the data is linearly separable.

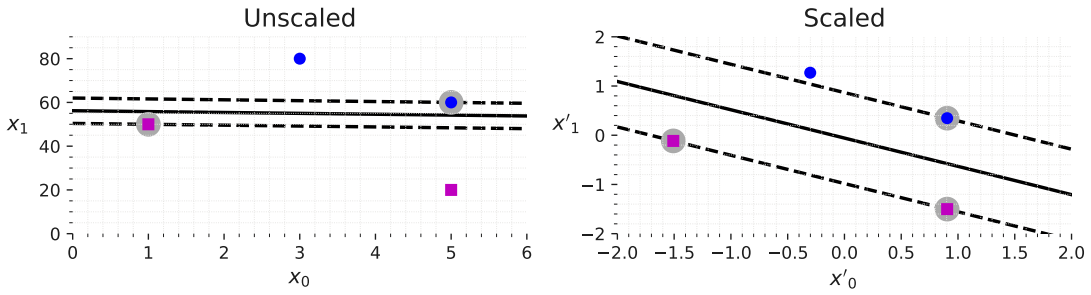

Figure 1.2: Sensitivity to feature scales.

- It is sensitive to **outliers**.

Outlier

A data point that differs significantly from other observations. An outlier may be due to a variability in the measurement, an indication of novel data, or it may be the result of experimental error.

Figure 1.3 shows the iris dataset with just one additional outlier: on the left, it is impossible to find a hard margin whereas on the right, the decision boundary ends up very different from the one we saw in Figure 1.1 without the outlier, and the model will probably not generalize as well.

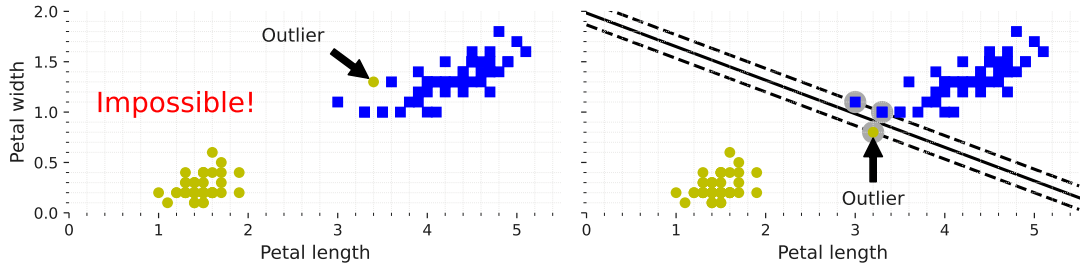

Figure 1.3: Hard margin sensitivity to outliers.

To avoid these aforementioned issues, we need to use a more flexible model. The idea is to **find a good balance between keeping the street as large as possible and limiting the margin violations** (i.e., instances that end up in the middle of the street or even on the wrong side). This is called **soft margin classification**.

When creating an SVM model using `sklearn`, you can specify several hyperparameters, including the regularisation hyperparameter `C`.

Hyperparameter `C`

Regularisation parameter. The strength of the regularisation is inversely proportional to `C` and must be **strictly positive**. The penalty is a squared ℓ_2 penalty. `float, default=1.0`

Setting it to a low value, then you end up with the model on the left of Figure 1.4. With a high value, you get the model on the right. As can be seen, reducing `C` makes the street larger, but it also leads to

more margin violations.

In other words, reducing `C` results in more instances supporting the street, so there's less risk of overfitting. But if you reduce it too much, then the model ends up underfitting, as seems to be the case here: the model with `C=100` looks like it will generalize better than the one with `C=1`.

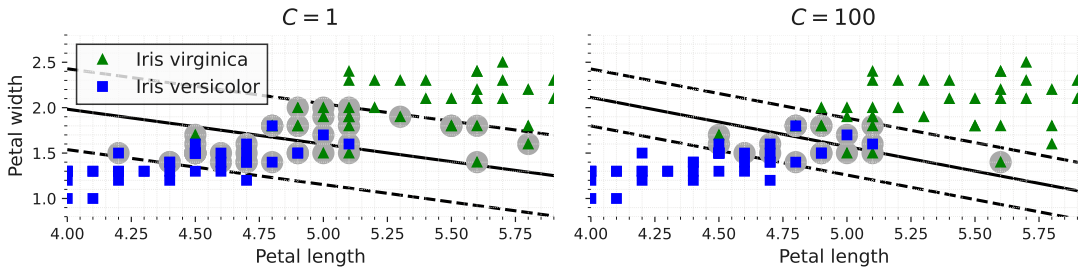

Figure 1.4: Large margin (left) v. fewer margin violations (right).

> If your SVM model is overfitting, you can try regularizing it by reducing `C`.

The following `sklearn` code loads the iris dataset and trains a linear SVM classifier to detect *Iris virginica* flowers. The pipeline first scales the features, then uses a `LinearSVC` with `C=1`:

C.R. 1 python

```
from sklearn.datasets import load_iris
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 2) # Iris virginica
svm_clf = make_pipeline(StandardScaler(),
LinearSVC(C=1, random_state=42))
svm_clf.fit(X, y)
```

The resulting model is represented on the left in Figure 1.4. Then, as usual, you can use the model to make predictions:

C.R. 2 python

```
X_new = [[5.5, 1.7], [5.0, 1.5]]
print(svm_clf.predict(X_new))
```

text

```
[ True False]
```

The first plant is classified as an *Iris virginica*, while the second is not. Let us look at the scores that the SVM used to make these predictions. These measure the signed distance between each instance and the decision boundary:

C.R. 3 python

```
print(svm_clf.decision_function(X_new))
```

```
[ 0.66163816 -0.22035761]
```
text

Unlike the `LogisticRegression` class, `LinearSVC` doesn't have a `predict_proba()` method to estimate the class probabilities. That said, if you use the SVC class instead of `LinearSVC`, and if you set its probability hyperparameter to `True`, then the model will fit an extra model at the end of training to map the SVM decision function scores to estimated probabilities.

Under the hood, this requires using 5-fold cross-validation to generate out-of-sample predictions for every instance in the training set, then training a `LogisticRegression` model, so it will slow down training considerably. After that, the `predict_proba()` and `predict_log_proba()` methods will be available.

1.3 Nonlinear SVM Classification

Although linear SVM classifiers are efficient and often work surprisingly well, many datasets are not even close to being linearly separable. One approach to handling nonlinear datasets is to add more features, such as polynomial features.

In some cases this can result in a linearly separable dataset. Consider the LHS plot in Figure 1.5: it represents a simple dataset with just one feature, x_1. This dataset is not linearly separable, as you can see. But adding a second feature $x_2 = x_1^2$, the resulting 2D dataset is perfectly linearly separable. To implement this idea using `sklearn`, you can create a pipeline containing a `PolynomialFeatures`

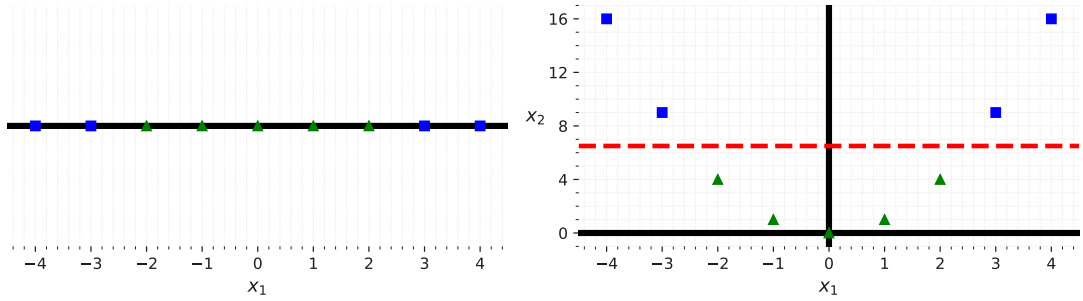

Figure 1.5: Adding features to make a dataset linearly separable.

transformer, followed by a `StandardScaler` and a `LinearSVC` classifier.

***Refresher:** Pipeline*

A series of interconnected data processing and modeling steps for streamlining the process of working with ML models.

Let's test this on the moons dataset, a toy dataset for binary classification in which the data points are shaped as two (2) interleaving crescent moons.

You can generate this dataset using the `make_moons()` function:

C.R. 4 python

```python
from sklearn.datasets import make_moons
from sklearn.preprocessing import PolynomialFeatures

X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

polynomial_svm_clf = make_pipeline(
    PolynomialFeatures(degree=3),
    StandardScaler(),
    LinearSVC(C=10, max_iter=10_000, random_state=42)
)
polynomial_svm_clf.fit(X, y)
```

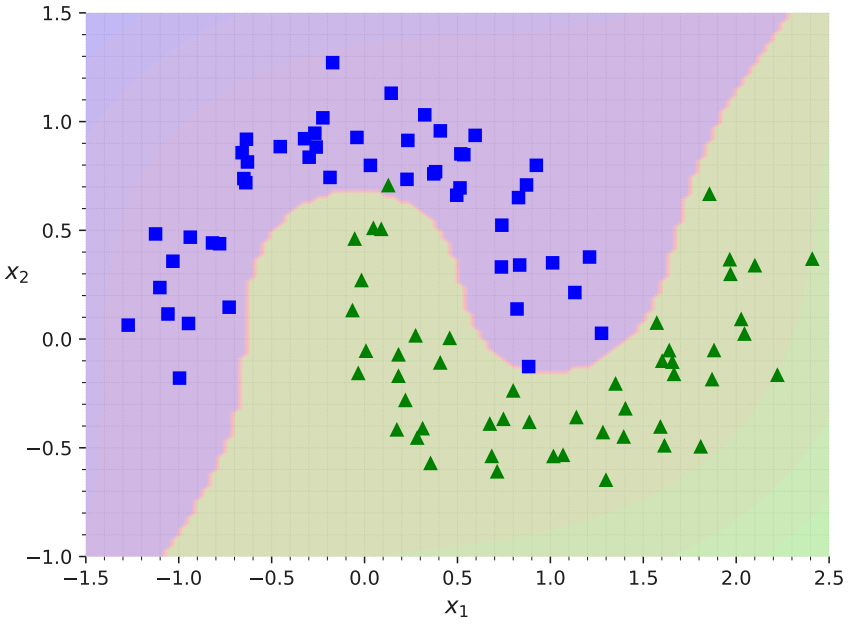

Figure 1.6: Linear SVM Classifier using polynomial features.

1.3.1 Polynomial Kernel

Adding polynomial features is simple to implement and can work great with all sorts of ML algorithms, and not just SVMs. That said, at a low polynomial degree this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.

Fortunately, when using SVMs you can apply a mathematical technique called the kernel trick. The kernel trick makes it possible to get the same result as if you had added many polynomial features, even with a very high degree, without actually having to add them.

Kernel Trick

A method used in SVMs to enable them to classify non-linear data using a linear classifier. By applying a kernel function, SVMs can implicitly map input data into a higher-dimensional

space where a linear separator (hyperplane) can be used to divide the classes. This mapping is computationally efficient because it avoids the direct calculation of the coordinates in this higher space.

This means there's no combinatorial explosion of the number of features.

This trick is implemented by the SVC class. Let's test it on the moons dataset:

C.R. 5

```python
from sklearn.svm import SVC

poly_kernel_svm_clf = make_pipeline(
    StandardScaler(),
    SVC(kernel="poly", degree=3, coef0=1, C=5)
)
poly_kernel_svm_clf.fit(X, y)
```

This code trains an SVM classifier using a third-degree polynomial kernel, represented on the left in Figure 1.7. On the right is another SVM classifier using a 10^{th} degree polynomial kernel. Obviously, if your model is overfitting, you might want to reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. The hyperparameter `coef0` controls how much the model is influenced by high-degree terms versus low-degree terms.

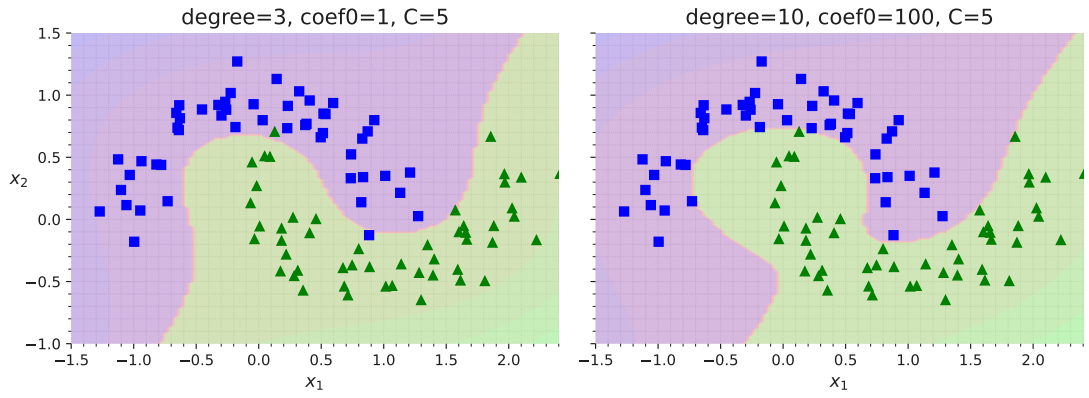

Figure 1.7: SVM classifiers with a polynomial kernel.

Although hyperparameters will generally be tuned automatically (e.g., using randomised search), it's good to have a sense of what each hyperparameter actually does and how it may interact with other hyperparameters: this way, you can narrow the search to a much smaller space.

1.3.2 Similarity Features

Another technique to tackle non-linear problems is to add features computed using a similarity function, which measures how much each instance resembles a particular landmark. For example, let's take the 1D dataset from earlier and add two landmarks to it at $x_1 = -2$ and $x_1 = 1$ (see the left plot

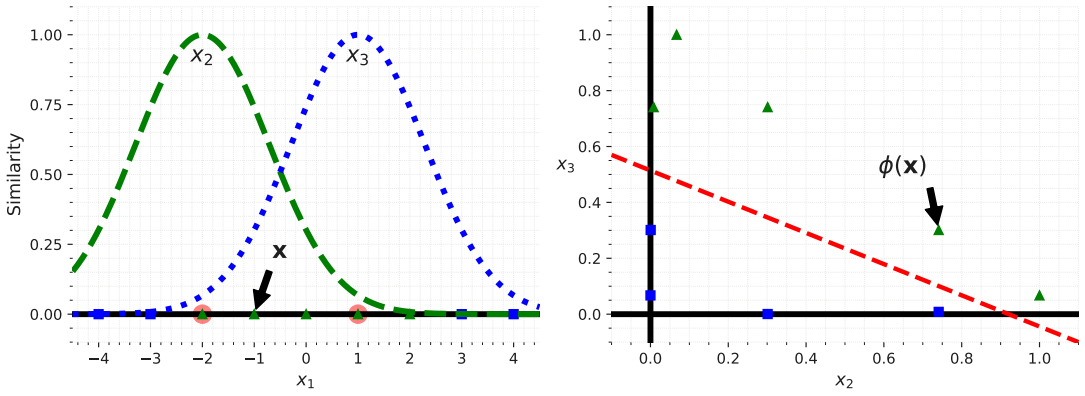

Figure 1.8: Similarity features using the Gaussian RBF.

in Figure 1.8). Next, we'll define the similarity function to be the Gaussian RBF with $\gamma = 0.3$. As it is a Gaussian function, it is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark).

$$\phi_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \,||\, \mathbf{x} - \ell \,||^2\right)$$

Now we are ready to compute the new features. For example, let's look at the instance $x_1 = -1$. It is located at a distance of 1 from the first landmark and 2 from the second landmark. Therefore, its new features are:

$$x_2 = e^{-0.3 \times 1} = 0.75 \qquad x_3 = e^{-0.3 \times 4} = 0.3$$

The plot on the right in Figure 1.8 shows the transformed dataset (dropping the original features). As you can see, it is now linearly separable.
You may wonder how to select the landmarks. The simplest approach is to create a landmark at the location of each and every instance in the dataset. Doing that creates many dimensions and thus increases the chances that the transformed training set will be linearly separable.

The downside is that a training set with m instances and n features gets transformed into a training set with m instances and m features (assuming you drop the original features).

> A very large training set, ends up with an equally large number of features.

1.3.3 Gaussian RBF Kernel

Just like the polynomial features method, the similarity features method can be useful with any ML algorithm, but it may be computationally expensive to compute all the additional features (especially on large training sets). Once again the kernel trick can be used here, making it possible to obtain a similar result as if you had added many similarity features, but without actually doing so. Let's try the SVC class with the Gaussian RBF kernel:

C.R. 6
python
```
rbf_kernel_svm_clf = make_pipeline(
    StandardScaler(),
    SVC(kernel="rbf", gamma=5, C=0.001)
)
```

C.R.7
```python
rbf_kernel_svm_clf.fit(X, y)
```

This model is represented at the bottom left in Figure 1.9. The other plots show models trained with different values of hyperparameters gamma (γ) and `C`.

Increasing gamma makes the bell-shaped curve narrower (see the LHS plots in Figure 1.9). As a result, each instance's range of influence is **smaller**. The decision boundary ends up being more irregular, wiggling around individual instances. Conversely, a small gamma value makes the bell-shaped curve wider: instances have a larger range of influence, and the decision boundary ends up smoother.

Therefore γ acts like a **regularisation hyperparameter**: if your model is overfitting, you should reduce γ; if it is underfitting, you should increase γ (similar to the `C` hyperparameter).

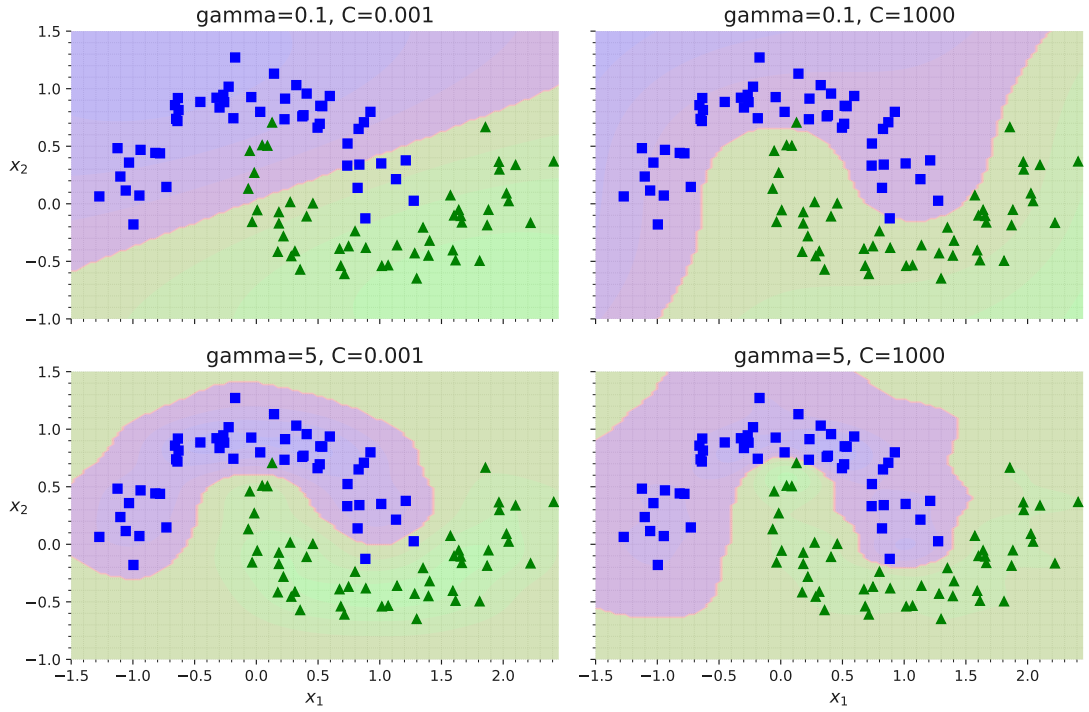

Figure 1.9: SVM classifiers using an RBF kernel.

Other kernels exist but are used much more rarely. Some kernels are specialized for specific data structures. String kernels are sometimes used when classifying text documents or DNA sequences (e.g., using the string subsequence kernel or kernels based on the Levenshtein distance).

> You need to choose the right kernel for the job. As a rule of thumb, you should always try the linear kernel first. The `LinearSVC` class is much faster than `SVC(kernel="linear")`, especially if the training set is very large. If it is not too large, you should also try kernelised SVMs, starting with the Gaussian RBF kernel; it often works really well. Then, if you have spare time and computing power, you can experiment with a few other kernels using hyperparameter search.

> If there are kernels specialized for your training set's data structure, make sure to give them a try too

1.4 SVM Regression

To use SVMs for regression instead of classification, the main idea is to tweak the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM regression tries to fit as many instances as possible on the street while limiting margin violations (i.e., instances off the street).

The width of the street is controlled by a hyperparameter, ϵ. Figure 1.10 shows two (2) linear SVM regression models trained on some linear data, one with a small margin (ϵ = 0.5) and the other with a larger margin (ϵ = 1.2). Reducing ϵ increases the number of support vectors, which regularises the

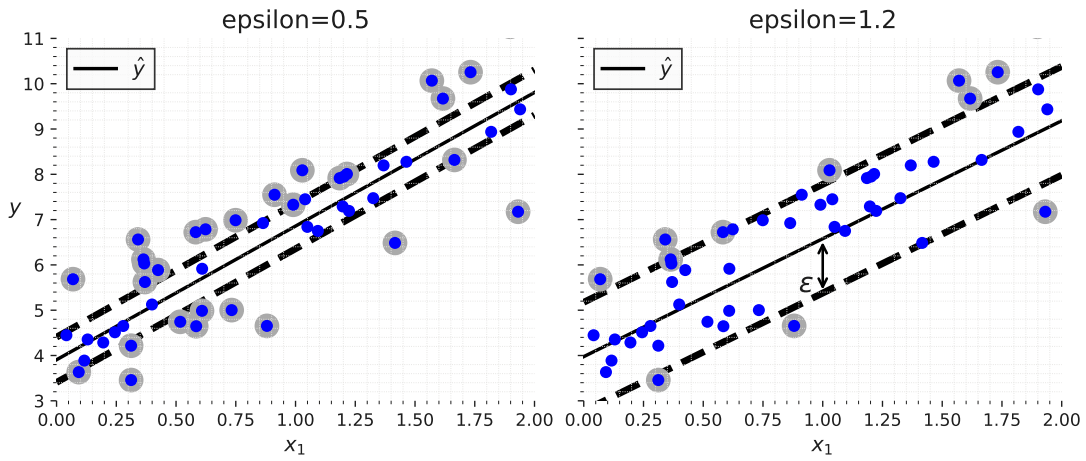

Figure 1.10: SVM regression.

model. Moreover, if you add more training instances within the margin, it will not affect the model's predictions; therefore, the model is said to be ϵ-insensitive.

You can use `sklearn`'s `LinearSVR` class to perform linear SVM regression. The following code produces the model represented on the left in Figure 1.10.

C.R. 8 python

```python
from sklearn.svm import LinearSVR

np.random.seed(42)
X = 2 * np.random.rand(50, 1)
y = 4 + 3 * X[:, 0] + np.random.randn(50)

svm_reg = make_pipeline(
    StandardScaler(),
    LinearSVR(epsilon=0.5, dual=True, random_state=42)
)
svm_reg.fit(X, y)
```

To tackle non-linear regression tasks, you can use a kernelized SVM model. Figure 1.11 shows SVM regression on a random quadratic training set, using a second-degree polynomial kernel. There is some regularisation in the left plot (i.e., a small `C` value), and much less in the right plot (i.e., a large `C` value). The following code uses `sklearn`'s SVR class (which supports the kernel trick) to produce the

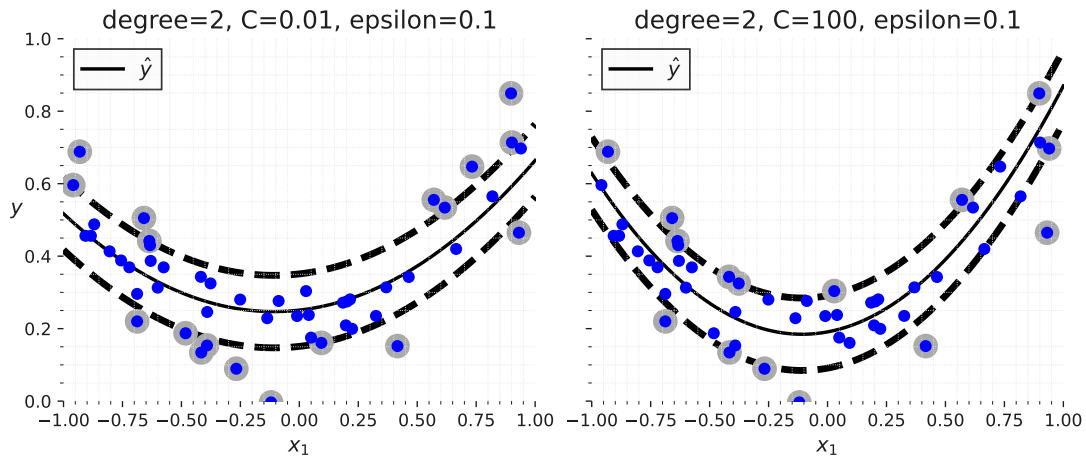

Figure 1.11: SVM regression using a second-degree polynomial kernel.

model represented on the left in Figure 1.11:

C.R. 9 python

```python
from sklearn.svm import SVR

# extra code – these 3 lines generate a simple quadratic dataset
np.random.seed(42)
X = 2 * np.random.rand(50, 1) - 1
y = 0.2 + 0.1 * X[:, 0] + 0.5 * X[:, 0] ** 2 + np.random.randn(50) / 10

svm_poly_reg = make_pipeline(
    StandardScaler(),
    SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1)
)

svm_poly_reg.fit(X, y)
```

The SVR class is the regression equivalent of the SVC class, and the `LinearSVR` class is the regression equivalent of the `LinearSVC` class. The `LinearSVR` class scales linearly with the size of the training set (just like the `LinearSVC` class), while the SVR class gets much too slow when the training set grows very large (just like the SVC class).

Chapter 2

Decision Trees

Table of Contents

2.1 Introduction

DT is a versatile ML algorithms that can perform both classification and regression tasks, and even multioutput tasks, capable of fitting complex datasets.

DTs are also the fundamental components of random forests, which are among the most powerful ML algorithms available today.

In this chapter we will start by discussing how to train, visualise, and make predictions with DTs. Then we will go through the CART training algorithm used by `sklearn`, and we will explore how to regularise trees and use them for regression tasks. Finally, we will discuss some of the limitations of DTs.

2.2 Training and Visualising Decision Trees

To understand DTs, let's build one and take a look at how it makes predictions. The following code trains a `DecisionTreeClassifier` on the iris dataset:

C.R. 1 python
```python
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris(as_frame=True)
X_iris = iris.data[["petal length (cm)", "petal width (cm)"]].values
y_iris = iris.target
tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X_iris, y_iris)
```

You can visualize the trained DT by first using the `export_graphviz()` function to output a graph definition file called `iris_tree.dot`:

C.R. 2 python
```python
from sklearn.tree import export_graphviz
export_graphviz(
tree_clf,
out_file="iris_tree.dot",
feature_names=["petal length (cm)", "petal width (cm)"],
class_names=iris.target_names,
rounded=True,
filled=True
)
```

If you are using a Jupyter Notebook to study, you can use `graphviz.Source.from_file()` to load and display the file inline:

C.R. 3 python
```python
from graphviz import Source
Source.from_file("iris_tree.dot")
```

Graphviz & DOT

Graphviz (short for Graph Visualization Software) is a package of open-source tools for creating graphs. It takes text input in DOT format, generates images.

DOT is a graph description language. DOT files are usually with .gv filename extension.

2.3 Making Predictions

Let's see how the tree represented in Figure 2.1 makes predictions.

Suppose you find an iris flower and you want to classify it based on its petals. You start at the root node (depth 0, at the top): this node asks whether the flower's petal length is smaller than 2.45 cm. If it is, then you move down to the root's left child node (depth 1, left). In this case, it is a leaf node (i.e., it does not have any child nodes), so it does not ask any questions: simply look at the predicted class for that node, and the DT predicts that your flower is an *Iris setosa* (`class=setosa`).

Now suppose you find another flower, and this time the petal length is greater than 2.45 cm. You again start at the root but now move down to its right child node (depth 1, right). This is not a leaf node, it's a **split node**, so it asks another question: is the petal width smaller than 1.75 cm? If it is, then

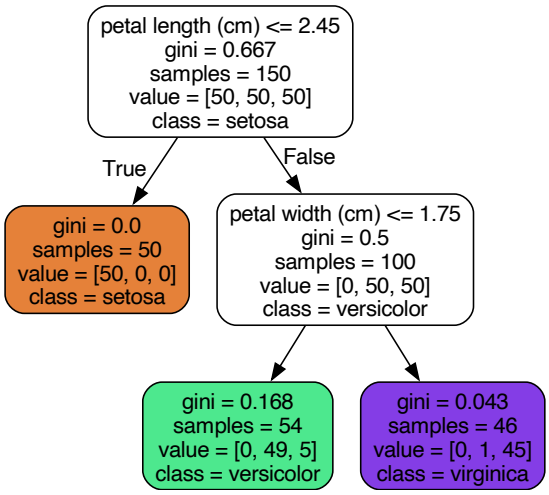

Figure 2.1

your flower is most likely an *Iris versicolor* (depth 2, left). If not, it is likely an *Iris virginica* (depth 2, right).
.

> One of the many qualities of DTs is that they require very little data preparation. In fact, they don't require feature scaling or centering at all.

A node's samples attribute counts how many training instances it applies to. For example, 100 training instances have a petal length greater than 2.45 cm (depth 1, right), and of those 100, 54 have a petal width smaller than 1.75 cm (depth 2, left). A node's value attribute tells you how many training instances of each class this node applies to.

For example, the bottom-right node applies to 0 Iris setosa, 1 Iris versicolor, and 45 Iris virginica. Finally, a node's gini attribute measures its **Gini impurity**: a node is "pure" (`gini=0`) if all training instances it applies to belong to the same class.

For example, since the depth-1 left node applies only to *Iris setosa* training instances, it is pure and its Gini impurity is 0. Eq. (2.1) shows how the training algorithm computes the Gini impurity G_i of the i^{th} node. The depth-2 left node has a Gini impurity of $1 - (0/54)2 - (49/54)2 - (5/54)2 \approx 0.168$.

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2 \tag{2.1}$$

where G_i is the Gini impurity of the i^{th} node, $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.

`sklearn` uses the CART algorithm, which produces only **binary trees**, meaning trees where split nodes always have exactly two children (i.e., questions only have yes/no answers).

However, other algorithms, such as ID3, can produce DTs with nodes that have more than two children.

Figure 2.2 shows this DT's decision boundaries. The thick vertical line represents the decision boundary of the root node (depth 0): petal length = 2.45 cm. Since the lefthand area is pure (only *Iris setosa*), it cannot be split any further. However, the righthand area is impure, so the depth-1 right node splits it at petal width = 1.75 cm (represented by the dashed line). Since `max_depth` was set to 2, the DT stops right there. If you set `max_depth` to 3, then the two depth-2 nodes would each add another decision boundary (represented by the two vertical dotted lines).

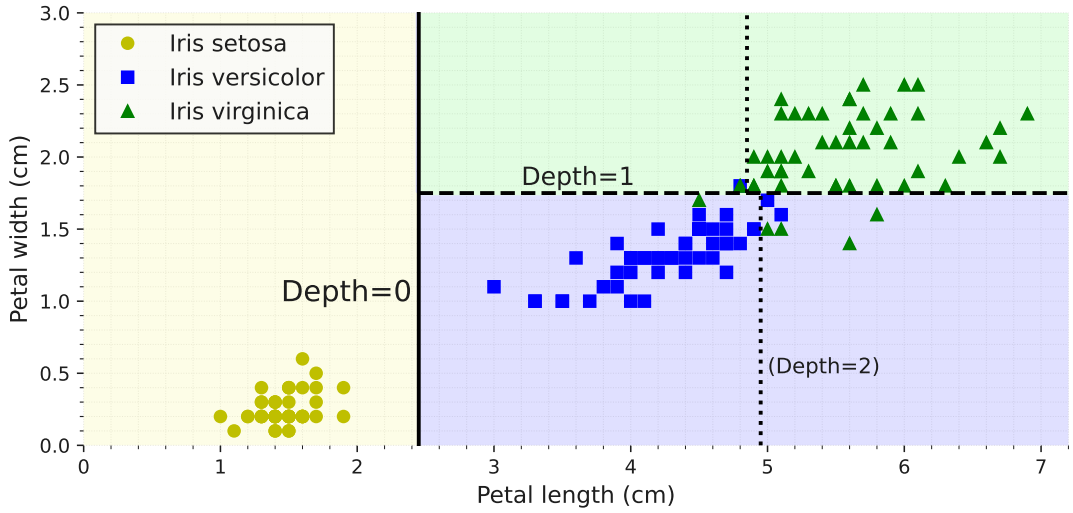

Figure 2.2: DT decision boundaries

The tree structure, including all the information shown in Figure 2.1, is available via the classifier's `tree_` attribute.

Type `help(tree_clf.tree_)` for details.

White v. Black Box

DTs are intuitive, and their decisions are easy to interpret. Such models are often called **white box** models. In contrast, random forests and neural networks are generally considered **black box** models. They make great predictions, and you can easily check the calculations that they performed to make these predictions, however, it is usually hard to explain in simple terms why the predictions were made.

For example, if a neural network says that a particular person appears in a picture, it is hard to know what contributed to this prediction: Did the model recognize that person's eyes? Their mouth? Their nose? Their shoes? Or even the couch that they were sitting on? Conversely, DTs provide nice, simple classification rules that can even be applied manually if need be (e.g., for flower classification). The field of interpretable ML aims at creating ML systems that can ex-

plain their decisions in a way humans can understand. This is important in many domains—for example, to ensure the system does not make unfair decisions.

2.4 Estimating Class Probabilities

A DT can also estimate the probability that an instance belongs to a particular class k. First, it traverses the tree to find the leaf node for this instance, and then it returns the ratio of training instances of class k in this node.

For example, suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide. The corresponding leaf node is the depth-2 left node, so the DT outputs the following probabilities: 0% for Iris setosa (0/54), 90.7% for Iris versicolor (49/54), and 9.3% for Iris virginica (5/54). And if you ask it to predict the class, it outputs Iris versicolor (class 1) because it has the highest probability. Let's check this:

C.R. 4 python
```python
print(tree_clf.predict_proba([[5, 1.5]]).round(3))
print(tree_clf.predict([[5, 1.5]]))
```

text
```
[[0.    0.907 0.093]]
[1]
```

Notice the estimated probabilities would be identical anywhere else in the bottom-right rectangle of Figure 2.1, for example, if the petals were 6 cm long and 1.5 cm wide.

2.5 The CART Training Algorithm

`sklearn` uses the Classification and Regression Tree (CART) algorithm to train DTs (also called growing trees). The algorithm works by first splitting the training set into two subsets using a single feature k and a threshold t_k (e.g., "petal length $\leq$ 2.45 cm").

How does it choose k and t_k? It searches for the pair (k, t_k) that produces the purest subsets, weighted by their size. Equation 2.2 gives the cost function that the algorithm tries to minimize.

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}} \quad \text{where} \quad \begin{cases} G_{\text{left/right}} & \text{measures the impurity} \\ m_{\text{left/right}} & \text{number of instances} \end{cases} \tag{2.2}$$

Once the CART algorithm successfully splits the training set in two, it splits the subsets using the same logic, then the sub-subsets, and so on, recursively. It stops recursing once it reaches the maximum depth (defined by the `max_depth` hyperparameter), or if it cannot find a split that will reduce impurity. A few other hyperparameters (described in a moment) control additional stopping conditions: `min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf`, and `max_leaf_nodes`.

CART algorithm is a **greedy algorithm**: it greedily searches for an optimum split at the top level, then repeats the process at each subsequent level. It does not check whether or not the split will lead to the lowest possible impurity several levels down. A greedy algorithm often

produces a solution that's reasonably good but not guaranteed to be optimal.

2.6 Gini Impurity or Entropy?

By default, the `DecisionTreeClassifier` class uses the **Gini impurity** measure, but you can select the entropy impurity measure instead by setting the criterion hyperparameter to entropy.

Entropy: A Measure of Disorder

The concept of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered. Entropy later spread to a wide variety of domains, including in Shannon's information theory, where it measures the average information content of a message and the entropy is zero when all messages are identical.

In ML, entropy is frequently used as an impurity measure: a set's entropy is zero when it contains instances of only one class. Equation 2.3 shows the definition of the entropy of the i^{th} node. For example, the depth-2 left node in Figure 2.1 has an entropy equal to:

$$-\frac{49}{54}\log_2\frac{49}{54}-\frac{5}{54}\log_2\frac{5}{54}\approx 0.445$$

And the general equation for entropy could be written as:

$$H_{\text{i}}=-\sum_{k=1}^{n}p_{\text{i,k}}\log_2 p_{\text{i,k}}\qquad\text{where } p_{\text{i,k}}\neq 0 \tag{2.3}$$

So, which one to use ? Gini impurity or entropy? Most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.

2.7 Regularization Hyperparameters

DTs make very few assumptions about the training data (as opposed to linear models, which assume that the data is linear, for example). If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely—indeed, most likely **overfitting** it.

Such a model is often called a non-parametric model, not because it does not have any parameters (it often has a lot) but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data.

In contrast, a parametric model, such as a linear model, has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of over-fitting

This increases the risk of underfitting.

To avoid overfitting the training data, you need to restrict the DT's freedom during training. As you know by now, this is called regularization. The regularization hyperparameters depend on the algorithm used, but generally you can at least restrict the maximum depth of the DT. In `sklearn`, this

is controlled by the `max_depth` hyperparameter. The default value is `None`, which means unlimited. Reducing `max_depth` will regularize the model and thus reduce the risk of overfitting.

The `DecisionTreeClassifier` class has a few other parameters that similarly restrict the shape of the DT:

`max_features` Maximum number of features that are evaluated for splitting at each node

`max_leaf_nodes` Maximum number of leaf nodes

`min_samples_split` Minimum number of samples a node must have before it can be split

`min_samples_leaf` Minimum number of samples a leaf node must have to be created

`min_weight_fraction_leaf` Same as `min_samples_leaf` but expressed as a fraction of the total number of weighted instances.

> Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters will regularize the model.

> Other algorithms work by first training the DT without restrictions, then pruning unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not statistically significant. Standard statistical tests, such as the χ^2 test (chi-squared test), are used to estimate the probability that the improvement is purely the result of chance (which is called the null hypothesis). If this probability, called the p-value, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. The pruning continues until all unnecessary nodes have been pruned.

Let's test regularization on the moons dataset, introduced previously. We'll train one DT without regularization, and another with `min_samples_leaf`=5. Here's the code; Figure 2.3 shows the decision boundaries of each tree:

C.R. 5 python
```python
from sklearn.datasets import make_moons

X_moons, y_moons = make_moons(n_samples=150, noise=0.2, random_state=42)

tree_clf1 = DecisionTreeClassifier(random_state=42)
tree_clf2 = DecisionTreeClassifier(min_samples_leaf=5, random_state=42)
tree_clf1.fit(X_moons, y_moons)
tree_clf2.fit(X_moons, y_moons)
```

The unregularized model on the left is clearly overfitting, and the regularized model on the right will probably generalize better. We can verify this by evaluating both trees on a test set generated using a different random seed:

C.R. 6 python
```python
X_moons_test, y_moons_test = make_moons(n_samples=1000, noise=0.2, random_state=43)
print(tree_clf1.score(X_moons_test, y_moons_test))
print(tree_clf2.score(X_moons_test, y_moons_test))
```

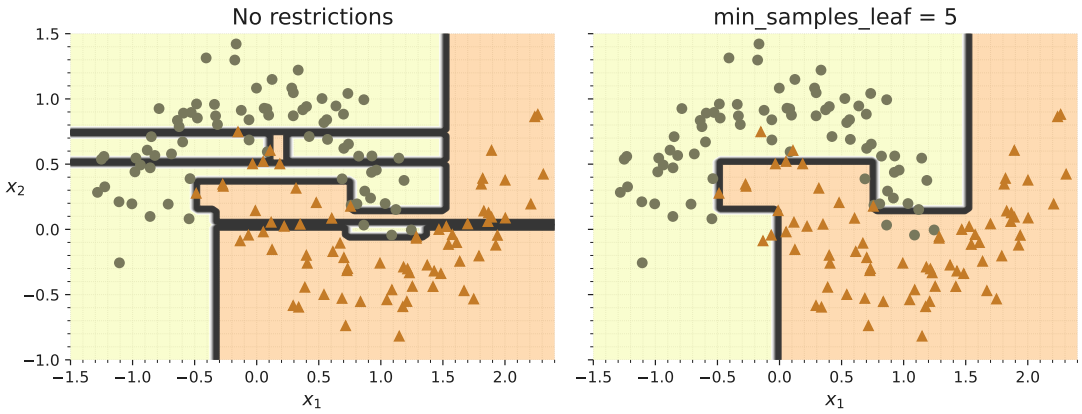

Figure 2.3: Decision boundaries of an unregularized tree (left) and a regularized tree (right)

```
0.898
0.92
```
text

Indeed, the second tree has a better accuracy on the test set.

2.8 Regression

DTs are also capable of performing regression tasks. Let's build a regression tree using `sklearn`'s `DecisionTreeRegressor` class, training it on a noisy quadratic dataset with `max_depth=2`:
The resulting tree is represented in Figure 2.4. This tree looks very similar to the classification tree built

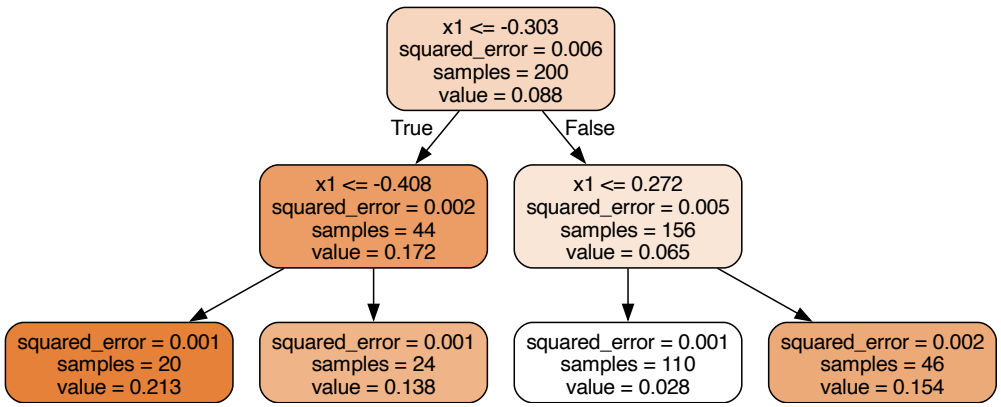

Figure 2.4: A DT for regression

earlier. The main difference is that instead of predicting a class in each node, it predicts a value. For example, suppose you want to make a prediction for a new instance with $x_1 = 0.2$. The root node asks whether $x_1 \leq 0.197$. Since it is not, the algorithm goes to the right child node, which asks whether $x_1 \leq 0.772$. Since it is, the algorithm goes to the left child node. This is a leaf node, and it predicts `value=0.111`. This prediction is the average target value of the 110 training instances associated with

this leaf node, and it results in a mean squared error equal to 0.015 over these 110 instances.

This model's predictions are represented on the left in Figure 2.5. If you set `max_depth=3`, you get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value. The CART algorithm

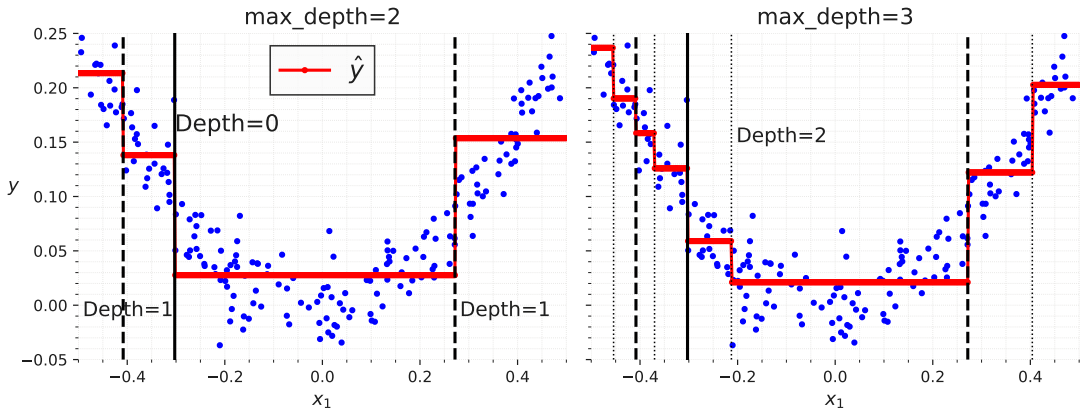

Figure 2.5: Predictions of two DT regression models

works as described earlier, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE. Equation 6-4 shows the cost function that the algorithm tries to minimize. Eq. (2.4) CART cost function for regression

$$J(k,\ t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}} \tag{2.4}$$

Just like for classification tasks, DTs are prone to overfitting when dealing with regression tasks. Without any regularization (i.e., using the default hyperparameters), you get the predictions on the left in Figure 2.6

These predictions are overfitting the training set very badly. Just setting `min_samples_leaf=10` results in a much more reasonable model, represented on the right in Figure 6-6.

2.9 Sensitivity to Axis Orientation

DTs have a lot going for them: they are relatively easy to understand and interpret, simple to use, versatile, and powerful. However, they do have a few limitations. First, as you may have noticed, DTs love orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to the orientation of data. For example, Figure 2.7 shows a simple linearly separable dataset: on the left, a DT can split it easily, while on the right, after the dataset is rotated by 45 degrees, the decision boundary looks unnecessarily convoluted. Although both DTs fit the training set perfectly, it is very likely that the model on the right will not generalize well.

One way to limit this problem is to scale the data, then apply a principal component analysis transformation. We will look at PCA in detail later, but for now you only need to know that it rotates the data in a way that reduces the correlation between the features, which often makes things easier for

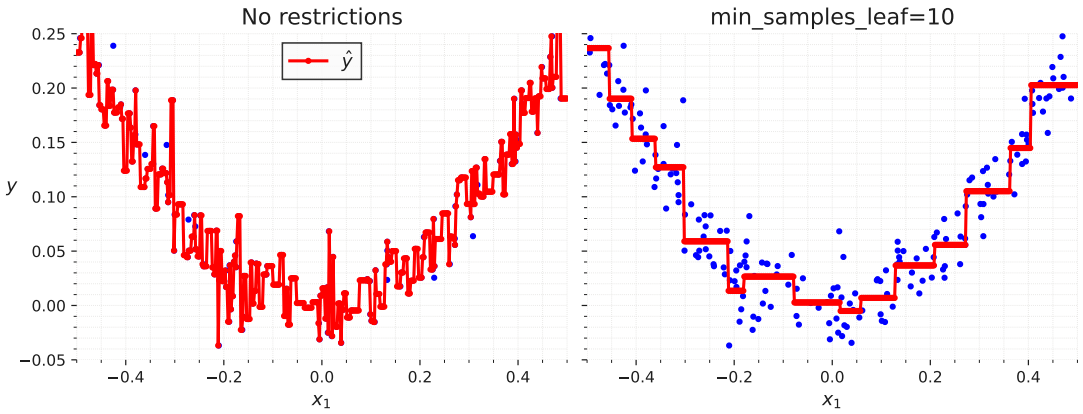

Figure 2.6: Predictions of an unregularized regression tree (left) and a regularized tree (right)

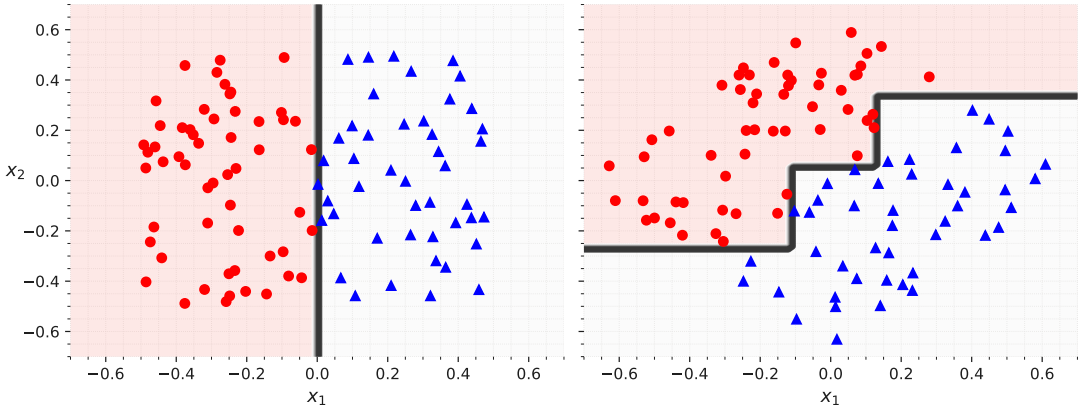

Figure 2.7: Sensitivity to training set rotation

trees. Let's create a small pipeline that scales the data and rotates it using PCA, then train a DecisionTreeClassifier on that data.

C.R. 7 python

```python
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

pca_pipeline = make_pipeline(StandardScaler(), PCA())
X_iris_rotated = pca_pipeline.fit_transform(X_iris)
tree_clf_pca = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf_pca.fit(X_iris_rotated, y_iris)
```

Figure 2.9 shows the decision boundaries of that tree: as you can see, the rotation makes it possible to fit the dataset pretty well using only one feature (1), which is a linear function of the original petal length and width.

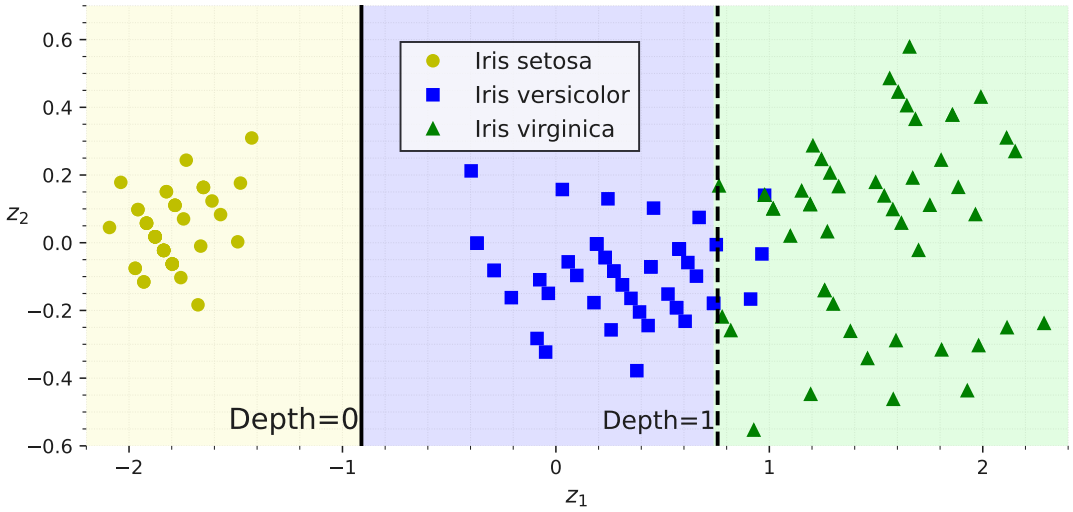

Figure 2.8: A tree's decision boundaries on the scaled and PCA-rotated iris dataset

2.10 DTs Have a High Variance

More generally, the main issue with DTs is that they have quite a **high variance**: small changes to the hyperparameters or to the data may produce very different models. In fact, since the training algorithm used by `sklearn` is stochastic—it randomly selects the set of features to evaluate at each node—even retraining the same DT on the exact same data may produce a very different model, such as the one represented in Figure 2.9 (unless you set the `random_state` hyperparameter). As you can see, it looks very different from the previous DT (shown in Figure 2.1. Luckily, by averaging predictions

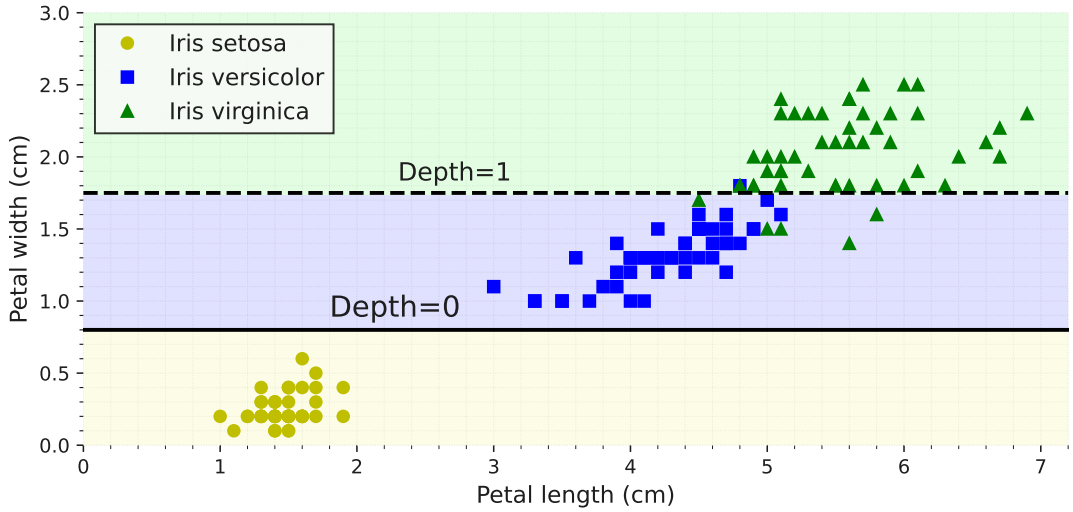

Figure 2.9: Retraining the same model on the same data may produce a very different model

over many trees, it's possible to reduce variance signiflicantly. Such an ensemble of trees is called a random forest, and it's one of the most powerful types of models available today, as you will see in the next chapter.

Chapter 3

Ensemble Learning and Random Forests

Table of Contents

Abstract

This chapter focuses on the ensemble learning methods where better decision making algorithms are made from small individual programs.

3.1 Introduction

Suppose you ask a complex question to millions of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer.

> This is called the wisdom of the crowd.

In a similar fashion, aggregating the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor.

> A group of predictors is called an **ensemble** and this technique, **ensemble learning**, and the learning method, **ensemble method**.

As an example of an ensemble method, you can train a group of decision tree classifiers, each on a different random subset of the training set. You can then obtain the predictions of all the individual trees, and the class that gets the most votes is the ensemble's prediction. Such an ensemble of decision trees is called a Random Forest (RF), and despite its simplicity, this is one of the most powerful ML algorithms available today. In this chapter we will examine the most popular ensemble methods, including:

- voting classifiers,
- bagging and pasting ensembles,
- RFs,
- boosting,
- and stacking ensembles.

3.1.1 Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about 80% accuracy. This may have a *logistic regression classifier*, an *SVM classifier*, a RF *classifier*, a *k-nearest neighbour classifier*, and perhaps a few more.

A simple way to create an even better classifier is to combine the predictions of each classifier:

The class that gets the most votes is the ensemble's prediction.

> This majority-vote classifier is called a **hard voting classifier**.

Interestingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a weak learner, meaning it does only slightly better than random guessing, the ensemble can still be a strong learner (achieving high accuracy), provided there are a sufficient number of weak learners in the ensemble and they are sufficiently diverse.

Let's discuss how this all works. Suppose you have a slightly biased coin that has a 51% chance of coming up heads and 49% chance of coming up tails. If you toss it 1,000 times, you will generally get more or less 510 heads and 490 tails, and hence a majority of heads.

If you do the calculation, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin, the higher the probability (e.g., with 10,000 tosses, the probability climbs over 97%). This is due to the **law of large numbers**: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%).

Law of Large Numbers (LLN)

A mathematical law states that the average of the results obtained from a large number of independent random samples converges to the true value, if it exists. More formally, the LLN states that given a sample of independent and identically distributed values, the sample mean converges to the true mean.

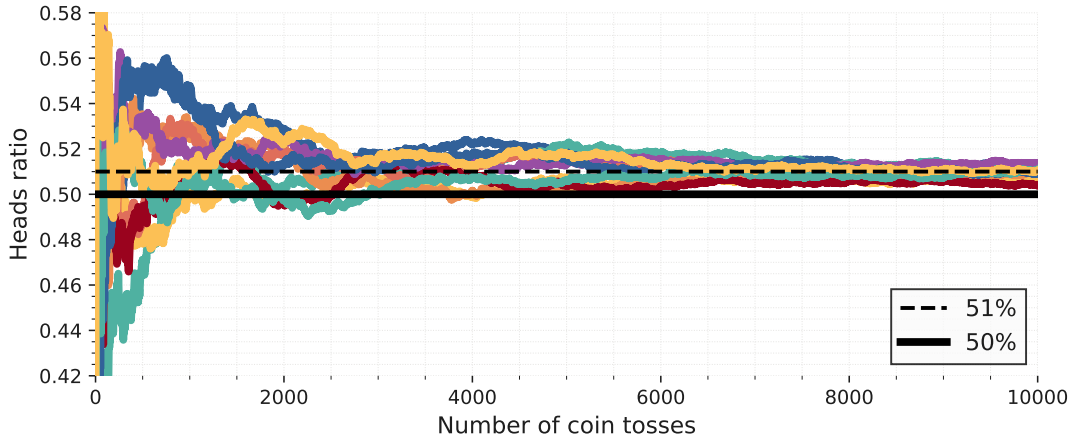

Figure 3.1: An example of a biased coin, where as the number of coin tosses increases the value of the will reach to 51%.

Extending this analogy to our case, assume you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing).

If you predict the majority voted class, you can hope for up to 75% accuracy.

However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case because they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

> Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

`sklearn` provides a `VotingClassifier` class which is intuitive: just give it a list of name/predictor pairs, and use it like a normal classifier. Let's try it on the moons dataset we used in SVM. We will load and split the moons dataset into a training set and a test set, then we'll create and train a voting classifier composed of three (3) diverse classifiers:

C.R.1
python
```
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
```

C.R. 2 python
```python
from sklearn.svm import SVC
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

When you fit a `VotingClassifier`, it clones every estimator and fits the clones. The original estimators are available via the `estimators` attribute, while the fitted clones are available via the `estimators_` attribute. If you prefer a `dict` rather than a list, you can use `named_estimators` or `named_estimators_` instead.

To begin, let's look at each fitted classifier's **individual** accuracy on the test set:

C.R. 3 python
```python
for name, clf in voting_clf.named_estimators_.items():
    print(name, "=", clf.score(X_test, y_test))
```

text
```
lr = 0.864
rf = 0.896
svc = 0.896
```

When you call the voting classifier's `predict()` method, it performs hard voting.

For example, the voting classifier predicts **class 1** for the first instance of the test set, because **two out of three classifiers** predict that class:

C.R. 4 python
```python
print(voting_clf.predict(X_test[:1]))
print([clf.predict(X_test[:1]) for clf in voting_clf.estimators_])
print(voting_clf.score(X_test, y_test))
```

text
```
[1]
[array([1]), array([1]), array([0])]
0.912
```

And we can look at the performance of the voting classifier on the test set which is `0.912`. As we can see, the voting classifier outperforms all the individual classifiers.

If all classifiers are able to estimate class probabilities (i.e., if they all have a `predict_proba()` method), then you can tell `sklearn` to predict the class with the highest class probability, averaged over all the individual classifiers.

This is called **soft voting**, which often achieves higher performance than hard voting because it gives more weight to highly confident votes. All you need to do is set the voting classifier's voting hyper-

parameter to `soft`, and ensure that all classifiers can estimate class probabilities. This is not the case for the SVC class by default, so you need to set its probability hyperparameter to `True`

> This will make the SVC class use cross-validation to estimate class probabilities, slowing down training, and it will add a `predict_proba()` method.

Let's try that:

C.R. 5

```python
voting_clf.voting = "soft"
voting_clf.named_estimators["svc"].probability = True
voting_clf.fit(X_train, y_train)
print(voting_clf.score(X_test, y_test))
```

```text
0.92
```

We reach 92% accuracy simply by using soft voting—not bad!

3.2 Bagging and Pasting

A way to get a diverse set of classifiers is to use very different training algorithms. An alternative approach is to use the same training algorithm for every predictor but train them on different random subsets of the training set.

When sampling is performed with replacement, this method is called **bagging** (short for bootstrap aggregating).

When sampling is performed without replacement, it is called **pasting**.

> Both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor.

Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. The aggregation function is typically the statistical mode for classification (i.e., the most frequent prediction, just like with a hard voting classifier), or the average for regression. Each individual predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance.

> Generally, the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.

> Predictors can all be trained in parallel, via different CPU cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons bagging and pasting are such popular methods: they scale very well.

3.2.1 Bagging and Pasting using sklearn

`sklearn` offers a simple classes for both bagging and pasting: the `BaggingClassifier` class (or `BaggingRegressor` for regression).

The code below trains an ensemble of 500 decision tree classifiers: each is trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`). The `n_jobs` parameter tells `sklearn` the number of CPU cores to use for training and predictions, and -1 tells `sklearn` to use all available cores:

C.R. 6 python

```python
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
```

> A `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a `predict_proba()` method), which is the case with decision tree classifiers.

Figure 3.2 compares the decision boundary of a single decision tree with the decision boundary of a bagging ensemble of 500 trees, both trained on the moons dataset. As can be seen, the ensemble's

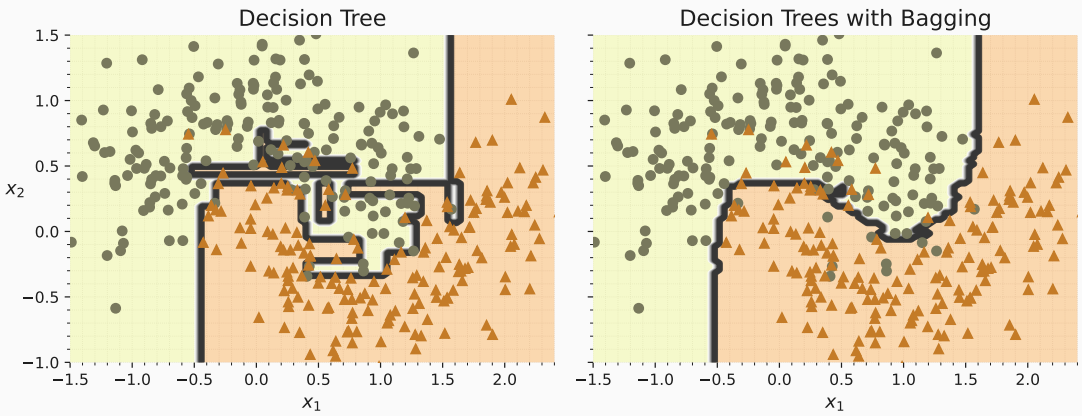

Figure 3.2: A single decision tree (left) versus a bagging ensemble of 500 trees (right)

predictions will likely generalize much better than the single decision tree's predictions: the ensemble has a comparable bias but a smaller variance.

> It makes roughly the same number of errors on the training set, but the decision boundary is less irregular

Bagging introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting; but the extra diversity also means that the predictors

end up being less correlated, so the ensemble's variance is reduced.

Overall, bagging often results in better models, which explains why it's generally preferred. But if you have spare time and CPU power, you can use cross-validation to evaluate both bagging and pasting and select the one that works best.

3.2.2 OoB Evaluation

With bagging, some training instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a `BaggingClassifier` samples m training instances with replacement (`bootstrap=True`), where m is the size of the training set. With this process, it can be shown mathematically that only about 63% of the training instances are sampled on average for each predictor.

The remaining 37% of the training instances that are not sampled are called OoB instances.

> Note that they are not the same 37% for all predictors.

A bagging ensemble can be evaluated using OoB instances, without the need for a separate validation set: indeed, if there are enough estimators, then each instance in the training set will likely be an OoB instance of several estimators, so these estimators can be used to make a fair ensemble prediction for that instance. Once you have a prediction for each instance, you can compute the ensemble's prediction accuracy (or any other metric). In `sklearn`, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic OoB evaluation after training. The following code demonstrates this. The resulting evaluation score is available in the `oob_score_` attribute:

C.R. 7

```python
bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            oob_score=True, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
print(bag_clf.oob_score_)
```

```text
0.896
```

According to this OoB evaluation, this `BaggingClassifier` is likely to achieve about 89.6% accuracy on the test set. Let's verify this:

C.R. 8

```python
from sklearn.metrics import accuracy_score

y_pred = bag_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```text
0.912
```

We get 92% accuracy on the test. The OoB evaluation was a bit too pessimistic, just over 2% too low. The OoB decision function for each training instance is also available as the `oob_decision_function_` attribute. As the base estimator has a `predict_proba()` method, the decision function returns the

class probabilities for each training instance. For example, the OoB evaluation estimates that the first training instance has a 67.6% probability of belonging to the positive class and a 32.4% probability of belonging to the negative class:

C.R. 9
```python
print(bag_clf.oob_decision_function_[:3])  # probas for the first 3 instances
```

```text
[[0.32352941 0.67647059]
 [0.3375     0.6625    ]
 [1.         0.        ]]
```

3.2.3 Random Patches and Random Subspaces

The `BaggingClassifier` class supports sampling the features as well. Sampling is controlled by two (2) hyper-parameters:

- `max_features`,
- `bootstrap_features`.

They work the same way as `max_samples` and bootstrap, but for feature sampling instead of instance sampling. Therefore, each predictor will be trained on a **random subset of the input features**.

This technique is particularly useful when you are dealing with highdimensional inputs (such as images), as it can considerably speed up training.

> Sampling both training instances and features is called the random patches method.

Keeping all training instances (by setting `bootstrap=False` and `max_samples=1.0`) but sampling features (by setting `bootstrap_features` to `True` and/or `max_features` to a value smaller than 1.0) is called the random subspaces method.

Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

3.3 Random Forests

As we have discussed, a RF is an **ensemble of decision trees**, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set. Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can use the `RandomForestClassifier` class, which is more convenient and optimised for decision trees (similarly, there is a `RandomForestRegressor` class for regression tasks). The following code trains a RF classifier with 500 trees, each limited to maximum 16 leaf nodes, using all available CPU cores:

C.R. 10
```python
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                 n_jobs=-1, random_state=42)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
```

With a few exceptions, a `RandomForestClassifier` has all the hyperparameters of a DecisionTreeClassifier (to control how trees are grown), plus all the hyperparameters of a `BaggingClassifier` to control the ensemble itself. The RF algorithm introduces **extra randomness** when growing trees. Instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features.

By default, it samples n features (where n is the total number of features). The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally giving an overall better model. So, the following `BaggingClassifier` is equivalent to the previous RandomForestClassifier:

C.R. 11
python
```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16),
    n_estimators=500, n_jobs=-1, random_state=42)
```

3.3.1 Extra-Trees

When you are growing a tree in a RF, at each node, only a random subset of the features is considered for splitting (as discussed earlier). It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular decision trees do). For this, simply set `splitter="random"` when creating a `DecisionTreeClassifier`.

A forest of such extremely random trees is called an extremely randomised trees (or extra-trees for short) ensemble.

> As with previous methods, this technique trades more bias for a lower variance.

It also makes extra-trees classifiers much faster to train than regular RFs, because finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree.

You can create an extra-trees classifier using `sklearn`'s `ExtraTreesClassifier` class. Its API is identical to the `RandomForestClassifier` class, except bootstrap defaults to `False`. Similarly, the `ExtraTreesRegressor` class has the same API as the `RandomForestRegressor` class, except bootstrap defaults to `False`.

> It is hard to tell in advance whether a `RandomForestClassifier` will perform better or worse than an `ExtraTreesClassifier`. Generally, the only way to know is to try both and compare them using cross-validation.

3.3.2 Feature Importance

Yet another great quality of RFs is that they make it easy to measure the relative importance of each feature. `sklearn` measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average, across all trees in the forest. More precisely, it is a weighted average, where each node's weight is equal to the number of training samples that are associated with

it.

`sklearn` computes this score automatically for each feature after training, then it scales the results so that the sum of all importances is equal to 1. You can access the result using the `feature_importances_` variable. The following code trains a `RandomForestClassifier` on the iris dataset and outputs each feature's importance.

C.R. 12 python

```python
from sklearn.datasets import load_iris

iris = load_iris(as_frame=True)
rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
rnd_clf.fit(iris.data, iris.target)
for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):
    print(round(score, 2), name)
```

text

```
0.11 sepal length (cm)
0.02 sepal width (cm)
0.44 petal length (cm)
0.42 petal width (cm)
```

It seems that the most important features are the petal length (44%) and width (42%), while sepal length and width are rather unimportant in comparison (11% and 2%, respectively):

Similarly, if you train a RF classifier on the MNIST dataset and plot each pixel's importance, you get the image represented in Figure 3.3. RFs are very handy to get a quick understanding of what features

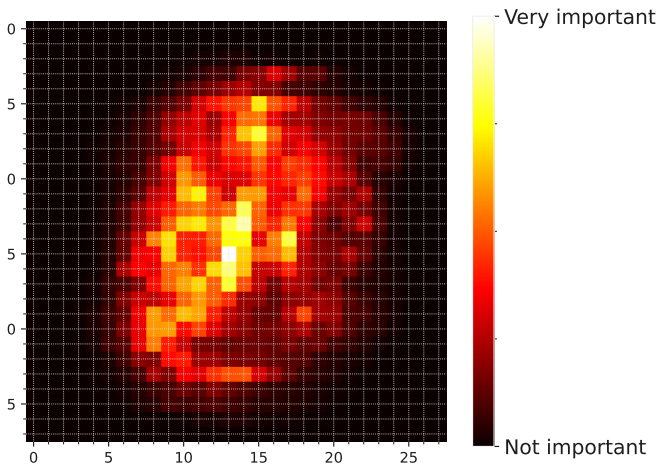

Figure 3.3: MNIST pixel importance (according to a RF classifier)

actually matter, in particular if you need to perform feature selection.

3.4 Boosting

Boosting (originally called hypothesis boosting) refers to any ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to **train**

predictors sequentially, each trying to correct its predecessor.

The general structure of it is as follows:

- Initially, a model is built using the training data.
- Subsequent models are then trained to address the mistakes of their predecessors.
- Boosting assigns weights to the data points in the original dataset.
  - **Higher weights:** Instances that were misclassified by the previous model receive higher weights.
  - **Lower weights:** Instances that were correctly classified receive lower weights.
- Training on weighted data: The subsequent model learns from the weighted dataset, focusing its attention on harder-to-learn examples (those with higher weights).
- This iterative process continues until the entire training dataset is accurately predicted, or A predefined maximum number of models is reached.

There are many boosting methods available, but by far the most popular are **AdaBoost** (short for adaptive boosting) and **gradient boosting**. Let's start with AdaBoost.

3.4.1 AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances which the predecessor **underfit**. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.

For example, when training an AdaBoost classifier, the algorithm first trains a base classifier (i.e., a decision tree) and uses it to make predictions on the training set. The algorithm then increases the relative weight of misclassified training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on. Figure 3.4 shows the decision boundaries of five (5) consecutive predictors on the moons dataset (in

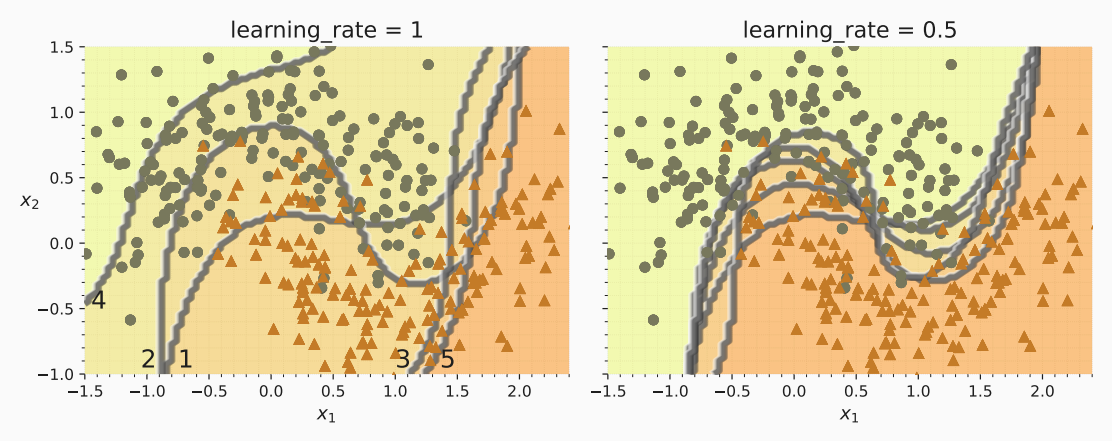

Figure 3.4: Decision boundaries of consecutive predictors.

this example, each predictor is a highly regularized SVM classifier with an RBF kernel). The first classifier gets many instances wrong, so their weights get boosted. The second classifier therefore does

a better job on these instances, and so on. The plot on the right represents the same sequence of predictors, except that the learning rate is halved (i.e., the misclassified instance weights are boosted much less at every iteration). As you can see, this sequential learning technique has some similarities with gradient descent, except that instead of tweaking a single predictor's parameters to minimise a cost function, AdaBoost adds predictors to the ensemble, gradually making it better.

Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.

> There is one important drawback to this sequential learning technique. training cannot be parallelized since each predictor can only be trained after the previous predictor has been trained and evaluated. Therefore, it does not scale as well as bagging or pasting.

Let's take a closer look at the AdaBoost algorithm.

Each instance weight $w^{(i)}$ is initially set to $1/m$. A first predictor is trained, and its weighted error rate r_1 is computed on the training set.

$$r_j = \frac{\sum_{i=1}^{m} w^{(i)} \quad \text{where} \quad \hat{y}_j^{(i)} \neq y^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

where $\hat{y}_j^{(i)}$ is the jth predictor's prediction for the ith instance. The predictor's weight α_j is then computed using Eq. (3.1), where η is the learning rate hyperparameter (which is 1 by default). The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to zero. However, if it is most often wrong (i.e., less accurate than random guessing), then its weight will be negative.

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j} \tag{3.1}$$

Next, the AdaBoost algorithm updates the instance weights, using Eq. (3.2), which boosts the weights of the misclassified instances.

$$w^{(i)} = \begin{cases} w^{(i)} & \text{if} \quad \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp \alpha_j & \text{if} \quad \hat{y}_j^{(i)} \neq y^{(i)} \end{cases} \tag{3.2}$$

Then all the instance weights are normalised by dividing it with $\sum_{i=1}^{m} w^{(i)}$).

Finally, a new predictor is trained using the updated weights, and the whole process is repeated: the new predictor's weight is computed, the instance weights are updated, then another predictor is trained, and so on. The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights α_j. The predicted class is the one that receives the majority of weighted votes.

`sklearn` uses a multiclass version of AdaBoost called SAMME (which stands for Stagewise Additive Modeling using a Multiclass Exponential loss function). When there are just two (2) classes, SAMME is equivalent to AdaBoost. If the predictors can estimate class probabilities (i.e., if they have a `predict_proba()` method), `sklearn` can use a variant of SAMME called SAMME.R (the R stands for "Real"), which relies on class probabilities rather than predictions and generally performs better.

The following code trains an AdaBoost classifier based on 30 decision stumps using `sklearn`'s AdaBoostClassifier class (as you might expect, there is also an AdaBoostRegressor class). A decision stump is a decision tree with `max_depth=1`, which in other words, a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class:

C.R. 13

```python
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=30,
    learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
```

Advantages	Disadvantages
Can effectively combine multiple weak classifiers to create a strong classifier with high accuracy	Can be sensitive to outliers and noisy data
Can handle complex datasets and capture intricate patters by iteratively adapting to difficult examples	Training process can be computationally expensive, especially dealing with large datasets.
By focusing on misclassified examples and adjusting sample weights, AdaBoost mitigates the risk of overfitting	Appropriate selection of weak classifiers and the number of hyperparameters are crucial for performance.
A versatile algorithm which can work with different types of base classifiers	Can struggle with imbalanced datasets.

Table 3.1: The advantages and disadvantages of AdaBoost.

> If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator.

3.4.2 Gradient Boosting

Another very popular boosting algorithm is gradient boosting. Just like AdaBoost, gradient boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the residual errors made by the previous predictor.

Let's go through a simple regression example, using decision trees as the base predictors; this is called gradient tree boosting, or gradient boosted regression trees (GBRT). First, let's generate a noisy quadratic

dataset and fit a `DecisionTreeRegressor` to it:

C.R.14
```python
import numpy as np
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3 * X[:, 0] ** 2 + 0.05 * np.random.randn(100)  # y = 3x² + Gaussian noise

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
```

Next, we'll train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

C.R.15
```python
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=43)
tree_reg2.fit(X, y2)
```

And then we'll train a third regressor on the residual errors made by the second predictor:

C.R.16
```python
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=44)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three (3) trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

C.R.17
```python
X_new = np.array([[-0.4], [0.], [0.5]])
print(sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3)))
```

```text
[0.49484029 0.04021166 0.75026781]
```

Fig. 3.5 represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

We can use `sklearn`'s `GradientBoostingRegressor` class to train GBRT ensembles more easily (there's also a `GradientBoostingClassifier` class for classification). Much like the `RandomForestRegressor` class, it has hyperparameters to control the growth of decision trees (e.g., `max_depth`, `min_samples_leaf`), as well as hyperparameters to control the ensemble training, such as the number of trees (`n_estimators`).

The following code creates the same ensemble as the previous one:

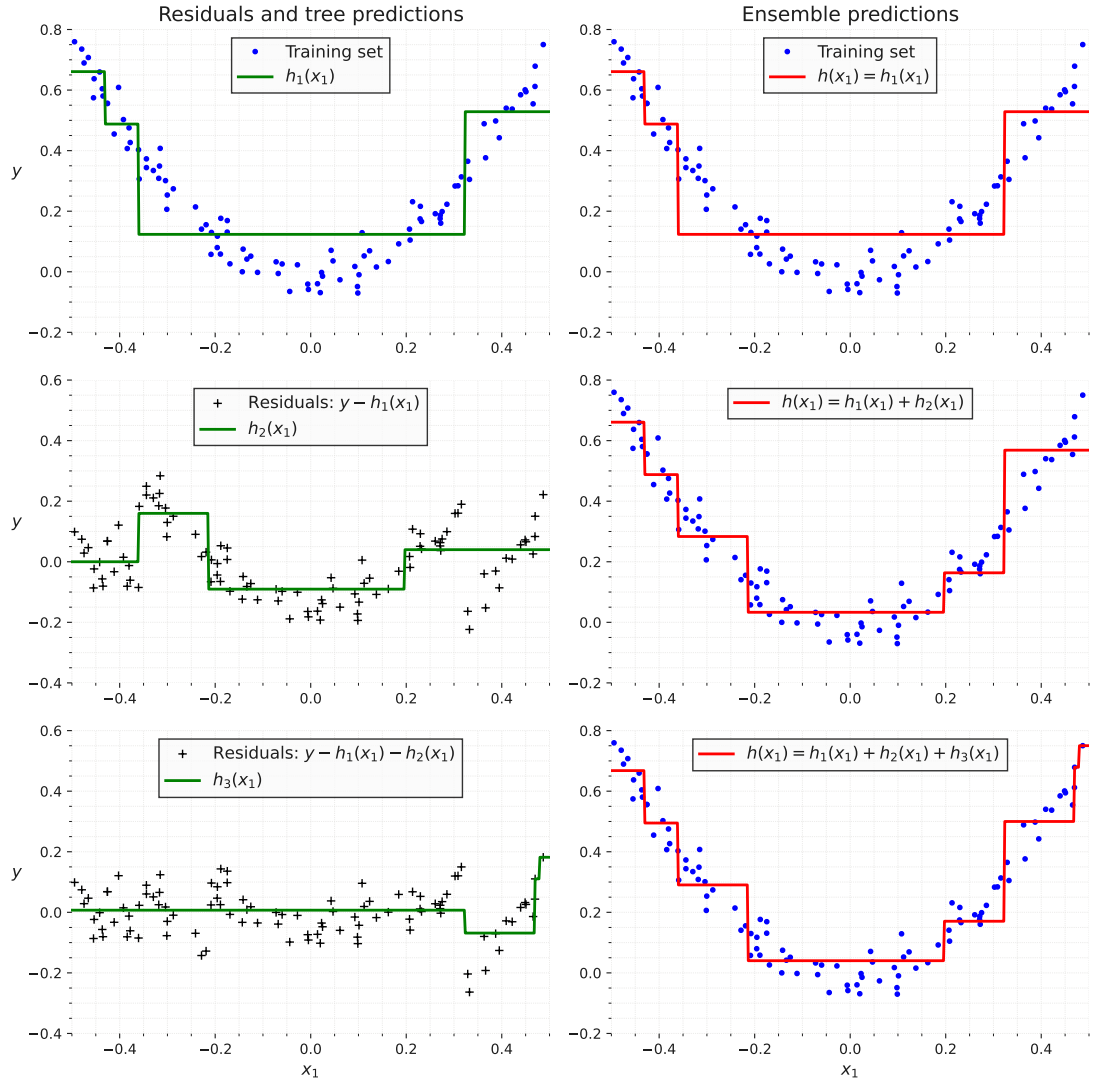

Figure 3.5: In this depiction of gradient boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

C.R. 18 python

```python
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
                                 learning_rate=1.0, random_state=42)
gbrt.fit(X, y)
```

The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as 0.05, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called **shrinkage** where **Fig.** 3.6 shows two GBRT ensembles trained with different hyperparameters: the one on the left does not have enough trees to fit the training set, while the one on the right has about the right amount. If we added more trees, the GBRT would start to overfit the training set. To find the optimal number of trees, you could perform cross-validation using `GridSearchCV` or `RandomizedSearchCV`, as usual, but there's a sim-

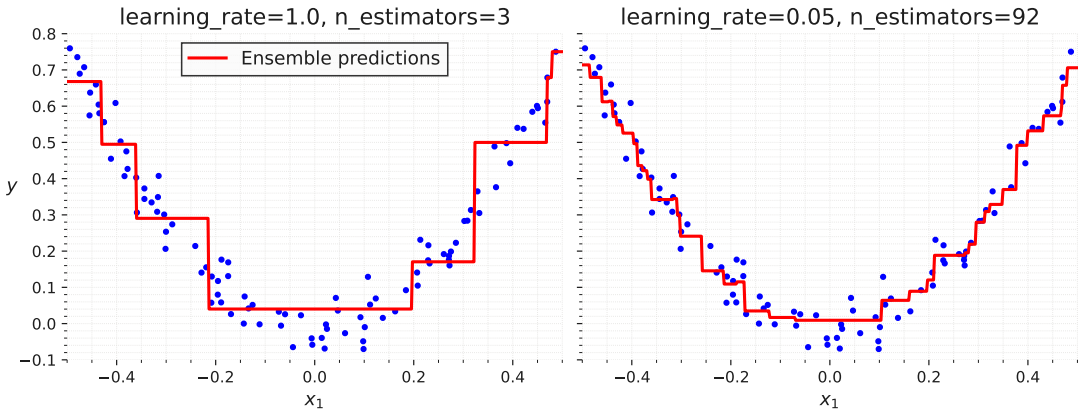

Figure 3.6: GBRT ensembles with not enough predictors (left) and just enough (right).

pler way: if you set the `n_iter_no_change` hyperparameter to an integer value, say 10, then the `GradientBoostingRegressor` will automatically stop adding more trees during training if it sees that the last 10 trees didn't help.

This is simply early stopping, but with a little bit of patience: it tolerates having no progress for a few iterations before it stops. Let's train the ensemble using early stopping:

C.R. 19
python
```
gbrt_best = GradientBoostingRegressor(
    max_depth=2, learning_rate=0.05, n_estimators=500,
    n_iter_no_change=10, random_state=42)
gbrt_best.fit(X, y)

```

If you set `n_iter_no_change` too low, training may stop too early and the model will underfit. But if you set it too high, it will overfit instead. We also set a fairly small learning rate and a high number of estimators, but the actual number of estimators in the trained ensemble is much lower, thanks to early stopping:

C.R. 20
python
```
print(gbrt_best.n_estimators_)
```

text
```
92
```

When `n_iter_no_change` is set, the `fit()` method automatically splits the training set into a smaller training set and a validation set: this allows it to evaluate the model's performance each time it adds a new tree. The size of the validation set is controlled by the `validation_fraction` hyperparameter, which is 10% by default. The `tol` hyperparameter determines the maximum performance improvement that still counts as negligible. It defaults to 0.0001. The `GradientBoostingRegressor` class also supports a subsample hyperparameter, which specifies the fraction of training instances to be used for training each tree.

For example, if `subsample=0.25`, then each tree is trained on 25% of the training instances, selected randomly. As you can probably guess by now, this technique trades a higher bias for a lower variance.

It also speeds up training considerably. This is called stochastic gradient boosting.

3.4.3 Histogram-Based Gradient Boosting

`sklearn` provides another GBRT implementation, optimised for large datasets: histogram based gradient boosting (HGB). It works by binning the input features, replacing them with integers. The number of bins is controlled by the `max_bins` hyperparameter, which defaults to 255 and cannot be set any higher than this. Binning can greatly reduce the number of possible thresholds that the training algorithm needs to evaluate. Moreover, working with integers makes it possible to use faster and more memory efficient data structures. And the way the bins are built removes the need for sorting the features when training each tree.

As a result, this implementation has a computational complexity of $\mathcal{O}(b \times m)$ instead of $\mathcal{O}(n \times m \log m)$, where b is the number of bins, m is the number of training instances, and n is the number of features. In practice, this means that HGB can train hundreds of times faster than regular GBRT on large datasets. However, binning causes a precision loss, which acts as a regularizer: depending on the dataset, this may help reduce overfitting, or it may cause underfitting.

`sklearn` provides two (2) classes for HGB:

1. `HistGradientBoostingRegressor`
2. `HistGradientBoostingClassifier`

They're similar to `GradientBoostingRegressor` and `GradientBoostingClassifier`, with a few notable differences:

- Early stopping is automatically activated if the number of instances is greater than 10,000. You can turn early stopping always on or always off by setting the `early_stopping` hyperparameter to True or False.
- Subsampling is not supported.
- `n_estimators` is renamed to `max_iter`.
- The only decision tree hyperparameters that can be tweaked are:
  - `max_leaf_nodes`,
  - `min_samples_leaf`,
  - and `max_depth`.

The HGB classes also have two (2) nice features: they support both categorical features and missing values. This simplifies preprocessing quite a bit. However, the categorical features must be represented as integers ranging from 0 to a number lower than `max_bins`. You can use an `OrdinalEncoder` for this.

For example, here's how to build and train a complete pipeline for the California housing dataset:

C.R. 21

```python
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
```

C.R. 22 python

```python
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OrdinalEncoder
hgb_reg = make_pipeline(
make_column_transformer((OrdinalEncoder(), ["ocean_proximity"]),
remainder="passthrough"),
HistGradientBoostingRegressor(categorical_features=[0], random_state=42)
)
hgb_reg.fit(housing, housing_labels)
```

The whole pipeline is just as short as the imports! No need for an imputer, scaler, or a one-hot encoder, so it's really convenient. Note that `categorical_features` must be set to the categorical column indices (or a Boolean array). Without any hyperparameter tuning, this model yields an RMSE of about 47,600, which is not too bad.

Implementations

Several other optimised implementations of gradient boosting are available in the Python ML ecosystem: in particular, XGBoost, CatBoost, and LightGBM. These libraries have been around for several years. They are all specialized for gradient boosting, their APIs are very similar to `sklearn`'s, and they provide many additional features, including GPU acceleration; you should definitely check them out! Moreover, the TensorFlow RFs library provides optimised implementations of a variety of RF algorithms, including plain RFs, extra-trees, GBRT, and several more.

3.5 Bagging v. Boosting

Similarities

Bagging and Boosting, both being the commonly used methods, have a universal similarity of being classified as ensemble methods. Here we will explain the similarities between them.

- Both are ensemble methods to get N learners from 1 learner.
- Both generate several training data sets by random sampling.
- Both make the final decision by averaging the N learners (or taking the majority of them i.e Majority Voting).
- Both are good at reducing variance and provide higher stability.

Differences

3.6 Stacking

The last ensemble method we will discuss in this chapter is called stacking (short for stacked generalization). It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation?

Bagging	Boosting
The simplest way of combining predictions that belong to the same type. classifiers to create a strong classifier with high accuracy	A way of combining predictions that belong to the different types.
Aim to decrease variance, not bias	Aim to decrease bias, not variance.
Each model receives equal weight.	Models are weighted according to their performance.
Each model is built independently.	New models are influenced by the performance of previously built models.
Different training data subsets are selected using row sampling with replacement and random sampling methods from the entire training dataset.	Iteratively train models, with each new model focusing on correcting the errors (misclassifications or high residuals) of the previous models
Bagging tries to solve the over-fitting problem.	Boosting tries to reduce bias.
In this base classifiers are trained in parallel.	In this base classifiers are trained sequentially.

Table 3.2: The advantages and disadvantages of AdaBoost.

To train the blender, you first need to build the blending training set. You can use `cross_val_predict()` on every predictor in the ensemble to get out-of-sample predictions for each instance in the original training set, and use these can be used as the input features to train the blender; and the targets can simply be copied from the original training set. Note that regardless of the number of features in the original training set (just one in this example), the blending training set will contain one input feature per predictor (three in this example). Once the blender is trained, the base predictors are retrained one last time on the full original training set.

It is actually possible to train several different blenders this way (e.g., one using linear regression, another using RF regression) to get a whole layer of blenders, and then add another blender on top of that to produce the final prediction. You may be able to squeeze out a few more drops of performance by doing this, but it will cost you in both training time and system complexity.

`sklearn` provides two classes for stacking ensembles: StackingClassifier and StackingRegressor. For example, we can replace the VotingClassifier we used at the beginning of this chapter on the moons dataset with a `StackingClassifier`:

C.R. 23
python
```python
from sklearn.ensemble import StackingClassifier
stacking_clf = StackingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42))
    ],
    final_estimator=RandomForestClassifier(random_state=43),

cv=5 # number of cross-validation folds
)
stacking_clf.fit(X_train, y_train)
```

For each predictor, the stacking classifier will call `predict_proba()` if available; if not it will fall back

to `decision_function()` or, as a last resort, call `predict()`. If you don't provide a final estimator, StackingClassifier will use `LogisticRegression` and StackingRegressor will use RidgeCV. If you evaluate this stacking model on the test set, you will find 92.8% accuracy, which is a bit better than the voting classifier using soft voting, which got 92%.

In conclusion, ensemble methods are versatile, powerful, and fairly simple to use. RFs, AdaBoost, and GBRT are among the first models you should test for most ML tasks, and they particularly shine with heterogeneous tabular data. Moreover, as they require very little preprocessing, they're great for getting a prototype up and running quickly. Lastly, ensemble methods like voting classifiers and stacking classifiers can help push your system's performance to its limits.

Chapter 4

Dimensionality Reduction

Table of Contents

Abstract

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five

4.1 Introduction

Many ML problems involve thousands or even millions of features for each training instance. Not only do all these features make training extremely slow, but they can also make it much harder to find a good solution, which we will see in the continuing parts of this chapter. This problem is often referred to as the curse of dimensionality[3][1].

Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images we worked previously. The pixels on the image borders are almost always white, so we could completely drop these pixels from the training set without losing much (if any) information.

As we saw in the previous chapter in Random Forest, we have confirmed these pixels are unimportant for the classification task. Additionally, two neighboring pixels are often highly correlated: if we merge

[1] The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience.

them into a single pixel (e.g., by taking the mean of the two pixel intensities), we will not lose much information.

Limits of Reduction

Reducing dimensionality does cause some **information loss**, similar to compressing an image to JPEG can degrade its quality, so even though it will speed up training, it may make your system perform slightly worse. It also makes your pipelines a bit more complex and thus harder to maintain.

Therefore, it is in our best interest to first try to train your system with the **original data** before considering using dimensionality reduction.

In some cases, reducing the dimensionality of the training data may filter out some noise and unnecessary details and thus result in higher performance, but in general it won't; it will just speed up training.

Apart from speeding up training, dimensionality reduction is also extremely useful for data visualisation. Reducing the number of dimensions down to two (or three) (2-3) makes it possible to plot a condensed view of a high-dimensional training set on a graph and often gain some important insights by visually detecting patterns, such as clusters[2]. Moreover, data visualisation is essential to communicate your conclusions to people who are not data scientists—in particular, decision makers who will use your results.

[2] a type of plot or mathematical diagram using Cartesian coordinates to display values for typically two variables for a set of data.

In this chapter of our lecture book we will first discuss the curse of dimensionality and get a sense of what goes on in high-dimensional space. Then we will consider the two main approaches to dimensionality reduction (projection and manifold learning), and we will go through three (3) of the most popular dimensionality reduction techniques:

1. Principal Component Analysis (PCA),
2. Random projection,
3. Locally Linear Embedding (LLE).

4.1.1 The Problems of Dimensions

We are used to living in three dimensions that our intuition fails us when we try to imagine a high-dimensional space. Even a basic 4D hypercube is incredibly hard to picture in our minds, let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.

> The more dimensions the data has, the less geometrically explainable it becomes.

Turns out many things behave very differently in high-dimensional space. As an example, Picking a random point in a unit square (a 1-by-1 square), will have only about a 0.4% chance of being located less than 0.001 from a border (in other words, it is very unlikely that a random point will be "extreme" along any dimension). However, if we do the same thin in a 10,000-dimensional unit hypercube, this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border.

n-dimensional Geometry

When dimensions become unruly the mathematics and our intuition becomes more divergent. There are many problems in mathematics where as the geometry becomes n-dimensional the results get even more complicated.

Here is another example: picking two (2) points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52. If you pick two random points in a 3D unit cube, the average distance will be roughly 0.66. But what about two points picked randomly in a 1,000,000-dimensional unit hypercube? The average distance, believe it or not, will be about 408.25.

This is counterintuitive: how can two points be so far apart when they both lie within the same unit hypercube? Well, **there's just plenty of space in high dimensions**. As a result, high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other. This also means that a new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions, since they will be based on much larger extrapolations.

> The more dimensions the training set has, the greater the risk of overfitting it.

In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features, which is significantly fewer than in the MNIST problem, all ranging from 0 to 1, you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

4.2 Main Approaches to Dimensionality Reduction

Before we dive into different dimensionality reduction algorithms, let's take a look at the two (2) main approaches to reducing dimensionality:

1. projection,
2. manifold learning.

4.2.1 Projection

In most real-world problems, training instances are **NOT** spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated. As a result, all training instances lie within a much lower-dimensional subspace of the high-dimensional space. This sounds very abstract, so let's look at an example.
In Fig. 4.1a you can see a 3D dataset represented by small spheres.
As you can see, almost all training instances lie close to a plane:

> this is a lower-dimensional (2D) subspace of the higher-dimensional (3D) space.

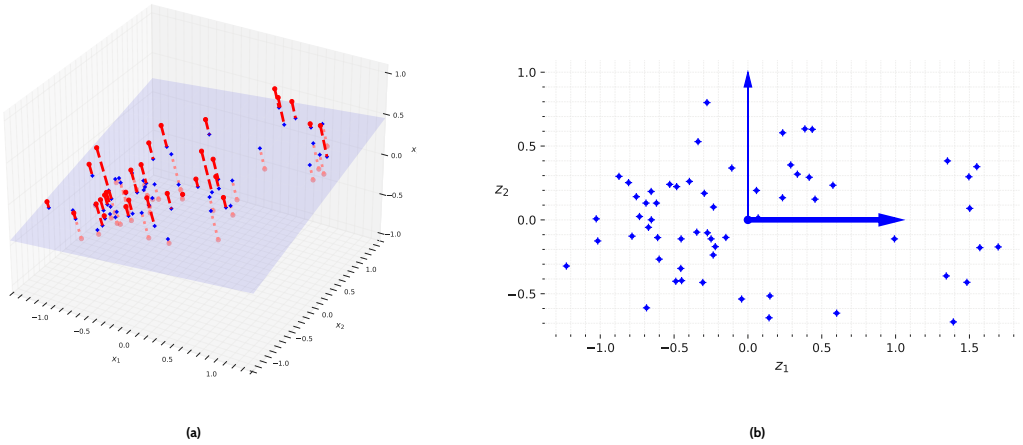

Figure 4.1: (a) A 3D dataset lying close to a 2D subspace (b) The new 2D dataset after projection

If we project every training instance perpendicularly onto this subspace (as represented by the short dashed lines connecting the instances to the plane), we get the new 2D dataset shown in Fig. 4.1b. We have just reduced the dataset's dimensionality from 3D to 2D. Note that the axes correspond to new features z_1 and z_2: they are the coordinates of the projections on the plane.

4.2.2 Manifold Learning

It is worth mentioning that, projection is not always the best approach to dimensionality reduction. In many cases the subspace may **twist** and **turn**, such as in the famous Swiss roll toy dataset[3] represented in Fig. 4.2.

[3] The Swiss roll is a toy dataset in `sklearn` that is commonly used for testing and demonstrating nonlinear dimensionality reduction algorithms. It consists of a set of points in three dimensions, arranged in a roll shape, such that the points on the roll are mapped to a two-dimensional plane in a nonlinear fashion

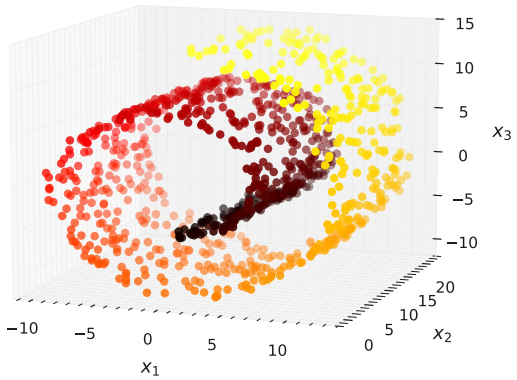

Figure 4.2: The Swiss roll dataset

Simply projecting onto a plane (e.g., by dropping the dimension x_3) would squash different layers of the Swiss roll together, as shown on the left side of Fig. 4.3. What we probably want instead is to unroll the Swiss roll to obtain the 2D dataset on the right side of Fig. 4.3.

The Swiss roll is an example of a 2D manifold. Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a d-dimensional manifold is a part of an n-dimensional space (where $d < n$) that locally resembles a d-dimensional hyperplane. In the case of the Swiss roll, $d = 2$ and $n = 3$.

> It locally resembles a 2D plane, but it is rolled in the 3rd dimension.

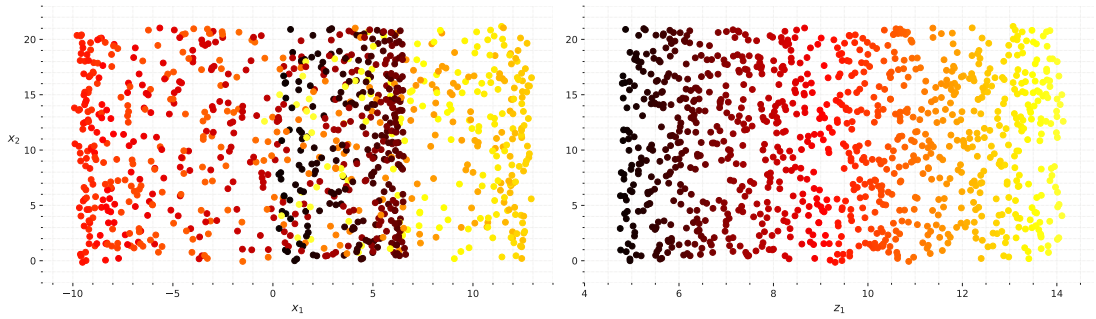

Figure 4.3: Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

Many dimensionality reduction algorithms work by modelling the manifold on which the training instances lie. this is called manifold learning [7]. It relies on the manifold assumption, also called the *manifold hypothesis*:

> *Most real-world high-dimensional datasets lie close to a much lower dimensional manifold.*

This assumption is very often **empirically**[4] observed.

[4] This means it is based on, concerned with, or verifiable by observation or experience rather than theory or pure logic

Once again, let us look back at the MNIST dataset: all handwritten digit images have some similarities. They are made of connected lines, the borders are white, and they are more or less centered. If you randomly generated images, only a ridiculously tiny fraction of them would look like handwritten digits. In other words, the degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you have if you are allowed to generate any image you want.

> These constraints tend to squeeze the dataset into a lower-dimensional manifold.

The manifold assumption is often accompanied by another **implicit** assumption:

> *The task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold*

For example, in the top row of Figure 8-6 the Swiss roll is split into two (2) classes: in the 3D space (on the left) the decision boundary would be fairly complex, but in the 2D unrolled manifold space (on the right) the decision boundary is a straight line.

However, this implicit assumption does not always hold. For example, in the bottom row of Figure 8-6, the decision boundary is located at $x_1 = 5$. This decision boundary looks very simple in the original

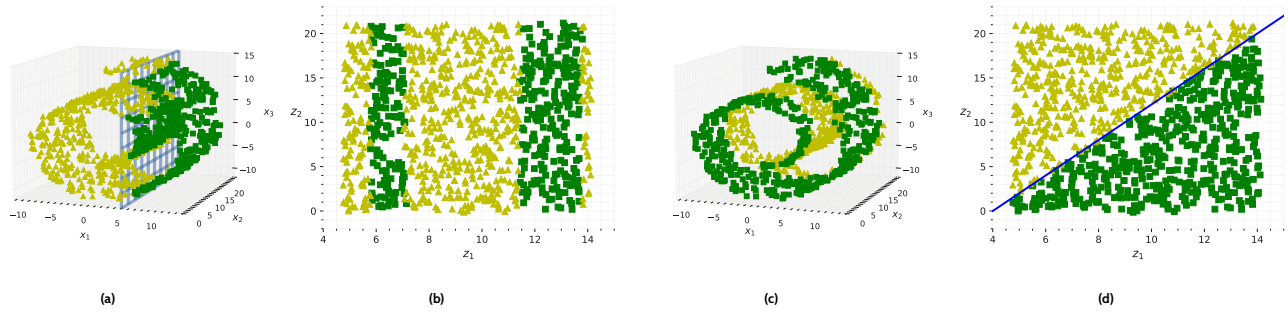

Figure 4.4: The decision boundary may not always be simpler with lower dimensions.

3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments).

> Reducing the dimensionality of your training set before training a model will usually speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset.

We now have a good sense of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold assumption holds. The rest of this chapter will go through some of the most popular algorithms for dimensionality reduction.

4.3 Principal Component Analysis (PCA)

[5] The method was invented in 1901 by Karl Pearson

Principal component analysis (PCA) is by far the most popular dimensionality reduction algorithm [5]. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it, just like in Fig 4.1a.

4.3.1 Preserving the Variance

Before we can project the training set onto a lower-dimensional hyperplane, we first need to choose the correct hyperplane. As an example, a simple 2D dataset is represented on the left in Fig. 4.5 along with three different axes (i.e., 1D hyperplanes).

On the right is the result of the projection of the dataset onto each of these axes. As you can see:

Top The projection onto the solid line preserves the maximum variance,

Bottom The projection onto the dotted line preserves very little variance

Middle The projection onto the dashed line preserves an intermediate amount of variance

It seems reasonable to select the *axis preserving the maximum amount of variance*, as it will most likely lose less information than the other projections.

> Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis.

This is idea behind PCA.

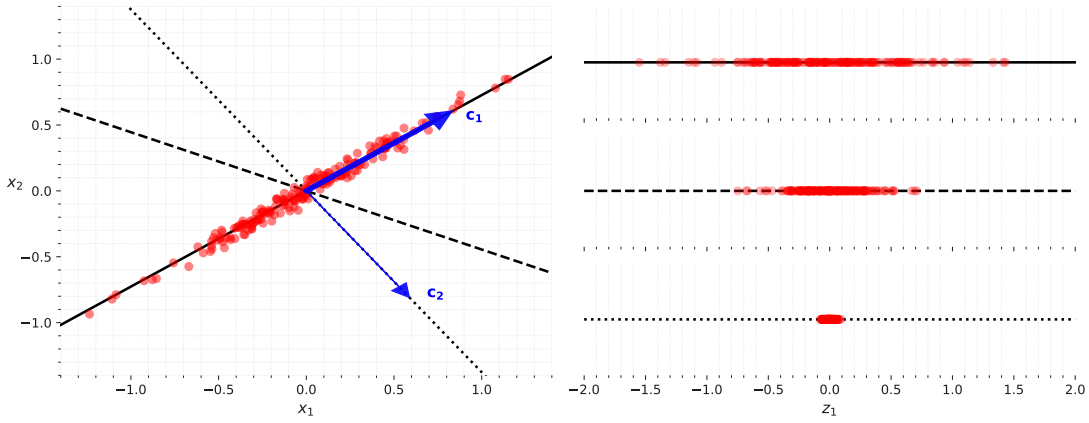

Figure 4.5: Selecting the subspace on which to project.

4.3.2 Principal Components

PCA identifies the axis accounting for the **largest amount of variance** in the training set.

in Fig. 4.5, it is the solid line.

It also finds a second axis, **orthogonal** to the first one, which accounts for the largest amount of the remaining variance. In this 2D example there is no choice: it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a 3rd axis, orthogonal to both previous axes, and a fourth, a fifth, and so on, as many axes as the number of dimensions in the dataset.

The i^{th} axis is called the ith Principal Component (PC) of the data. In Fig. 4.5, the first PC is the axis on which vector c_1 lies, and the second PC is the axis on which vector c_2 lies.

In Fig. 4.1a the first two PCs are on the projection plane, and the third PC is the axis orthogonal to that plane. After the projection, in Fig. 4.1b, the first PC corresponds to the z_1 axis, and the second PC corresponds to the z_2 axis.

> For each PC, PCA finds a zero-centered unit vector pointing in the direction of the PC. As two opposing unit vectors lie on the same axis, the direction of the unit vectors returned by PCA is not stable: if you perturb the training set slightly and run PCA again, the unit vectors may point in the opposite direction as the original vectors. However, they will generally still lie on the same axes. In some cases, a pair of unit vectors may even rotate or swap (if the variances along these two axes are very close), but the plane they define will generally remain the same.

So how can we find the principal components of a training set?

Luckily, there is a standard matrix factorisation technique called Singular Value Decomposition (SVD) which decomposes the training set matrix $\mathbf{X}$ into the matrix multiplication of three (3) matrices $\mathbf{U\Sigma V}^T$, where $\mathbf{V}$ contains the unit vectors that define all the principal components that you are looking for, as shown below:

$$\mathbf{V} = \begin{bmatrix} c_1 & c_2 & \cdots & c_n \end{bmatrix}$$

The following Python code uses NumPy's `svd()` function to obtain all the principal components of the 3D training set represented in Fig. 4.1a, then it extracts the two unit vectors that define the first two PCs:

C.R. 1
python
```
import numpy as np

X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt[0]
c2 = Vt[1]
```

> PCA assumes that the dataset is centered around the origin. As we will see, `sklearn`'s PCA classes take care of centering the data for us. If we were to implement PCA ourselves (as in the preceding example), or if we used other libraries, we shouldn't forget to center the data first.

4.3.3 Downgrading Dimensions

Once we identified all the PCs, we can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible. For example, in Fig 4.1a the 3D dataset is projected down to the 2D plane defined by the first two (2) principal components, preserving a large part of the dataset's variance. As a result, the 2D projection looks very much like the original 3D dataset.

To project the training set onto the hyperplane and obtain a reduced dataset $\mathbf{X}_{d-proj}$ of dimensionality d, compute the matrix multiplication of the training set matrix $\mathbf{X}$ by the matrix W_d, defined as the matrix containing the first d columns of $\mathbf{V}$:

$$\mathbf{X}_{d-proj} = \mathbf{X}\mathbf{W}_d$$

The following Python code projects the training set onto the plane defined by the first two principal components:

C.R. 2
python
```
W2 = Vt[:2].T
X2D = X_centered @ W2
```

We now know how to reduce the dimensionality of any dataset by projecting it down to any number of dimensions, while preserving as much variance as possible.

Using sklearn

`sklearn`'s `PCA` class uses SVD to implement PCA, just like we did earlier in this chapter. The following code applies PCA to reduce the dimensionality of the dataset down to two (2) dimensions:

> `sklearn`'s `PCA` also automatically takes care of centering the data.

C.R. 3
python
```python
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X2D = pca.fit_transform(X)
```

After fitting the PCA transformer to the dataset, its `components_` attribute holds the transpose of $\mathbf{W}_d$: it contains one row for each of the first d principal components.

Explained Variance Ratio

Another useful piece of information is the explained variance ratio of each principal component, available via the `explained_variance_ratio_` variable. The ratio indicates the proportion of the dataset's variance that lies along each principal component. For example, let's look at the explained variance ratios of the first two components of the 3D dataset represented in Fig. 4.1a:

C.R. 4
python
```python
print(pca.explained_variance_ratio_)
```

text
```
[0.17974135 0.1177597 ]
```

This output tells us that about 76% of the dataset's variance lies along the first PC, and about 15% lies along the second PC. This leaves about 9% for the third PC, so it is reasonable to assume that the third PC probably carries little information.

4.3.4 The Right Number of Dimensions

Instead of randomly choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance, for example, 95%.

> An exception to this rule, of course, is if you are reducing dimensionality for data visualization, in which case you will want to reduce the dimensionality down to 2 or 3.

The following code loads and splits the MNIST dataset and performs PCA without reducing dimensionality. It then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

C.R. 5
python
```python
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False, parser="auto")
X_train, y_train = mnist.data[:60_000], mnist.target[:60_000]
X_test, y_test = mnist.data[60_000:], mnist.target[60_000:]

pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1  # d equals 154
```

We could then set `n_components=d` and run PCA again, but there's a better option. Instead of specifying the number of principal components you want to preserve, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

C.R. 6 python
```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

The actual number of components is determined during training, and it is stored in the `n_components_` attribute:

C.R. 7 python
```
print(pca.n_components_)
```

text
```
154
```

A different option is to plot the explained variance as a function of the number of dimensions which you can see in Fig. 4.6. There will usually be an elbow in the curve, where the explained variance stops growing fast. In this case, we can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.

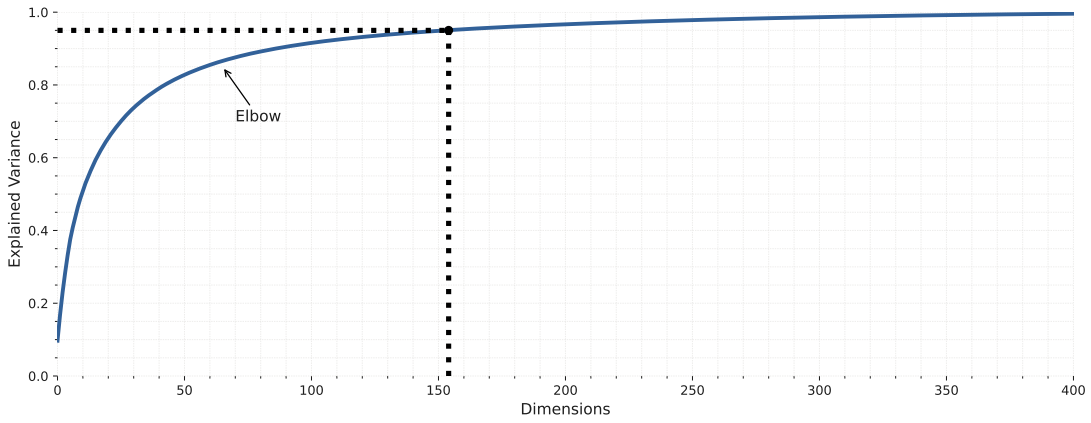

Figure 4.6: Explained variance as a function of the number of dimensions.

Lastly, if you are using dimensionality reduction as a pre-processing step for a supervised learning task, then you can tune the number of dimensions as you would any other hyperparameter.

For example, the following code example creates a two-step pipeline, first reducing dimensionality using PCA, then classifying using a random forest. Next, it uses RandomizedSearchCV to find a good combination of hyperparameters for both PCA and the random forest classifier. This example does a quick search, tuning only two (2) hyperparameters, training on just 1,000 instances, and running for just 10 iterations:

C.R. 8 python
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline

```

C.R. 9 python
```python
clf = make_pipeline(PCA(random_state=42),
                    RandomForestClassifier(random_state=42))
param_distrib = {
    "pca__n_components": np.arange(10, 80),
    "randomforestclassifier__n_estimators": np.arange(50, 500)
}
rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3,
                                random_state=42)
rnd_search.fit(X_train[:1000], y_train[:1000])


```

Let's look at the best hyperparameters found:

C.R. 10 python
```python
print(rnd_search.best_params_)
```

text
```
{'randomforestclassifier__n_estimators': 475, 'pca__n_components': 57}
```

It's interesting to see how low the optimal number of components is: we reduced a 784-dimensional dataset to just 23 dimensions. This is tied to the fact that we used a random forest, which is a pretty powerful model. If we used a linear model instead, such as an `SGDClassifier`, the search would find that we need to preserve more dimensions.

4.3.5 PCA for Compression

After dimensionality reduction, the training set takes up much less space. For example, after applying PCA to the MNIST dataset while preserving 95% of its variance, we are left with 154 features, instead of the original 784 features. So the dataset is now less than 20% of its original size, and we only lost 5% of its variance. This is a reasonable compression ratio, and it's easy to see how such a size reduction would speed up a classification algorithm tremendously.

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. This won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be close to the original data. The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the **reconstruction error** [11].
The `inverse_transform()` method lets us decompress the reduced MNIST dataset back to 784 dimensions:

C.R. 11 python
```python
X_recovered = pca.inverse_transform(X_reduced)
```

Fig. 4.7 shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact. The equation for the inverse transformation is shown below.

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d-\text{proj}}\mathbf{W}_d^T$$

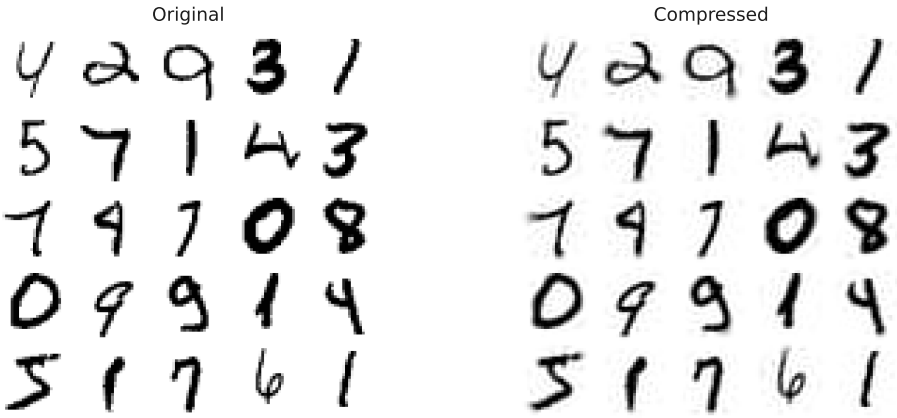

Figure 4.7: MNIST compression that preserves 95% of the variance

4.3.6 Randomized PCA

If you set the `svd_solver` hyperparameter to `randomized`, `sklearn` uses a stochastic algorithm called randomised PCA that quickly finds an approximation of the first d principal components. Its computational complexity is:

$$\mathcal{O}\left(m \times d^2\right) + \mathcal{O}\left(d^3\right) \qquad \text{Instead of} \qquad \mathcal{O}\left(m \times n^2\right) + \mathcal{O}\left(n^3\right)$$

for the full SVD approach, therefore it is dramatically faster than full SVD when d is much smaller than n:

C.R. 12
python
```
rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```

> By default, `svd_solver` is set to `"auto"`: `sklearn` automatically uses the randomised PCA algorithm if $max(m, n) > 500$ and `n_components` is an integer smaller than 80% of $min(m, n)$, or else it uses the full SVD approach. So the preceding code would use the randomized PCA algorithm even if you removed the `svd_solver="randomized"` argument, as $154 < 0.8 \times 784$.
>
> If we want to force `sklearn` to use full SVD for a slightly more precise result, you can set the `svd_solver` hyperparameter to `"full"`.

4.3.7 Incremental PCA

One problem with the preceding implementations of PCA is that they require the whole training set to fit in memory in order for the algorithm to run. Fortunately, incremental PCA algorithms have been developed that allow you to split the training set into mini-batches and feed these in one mini-batch at a time. This is useful for large training sets and for applying PCA online (i.e., on the fly, as new instances arrive).

The following code splits the MNIST training set into 100 mini-batches (using NumPy's `array_split()` function) and feeds them to `sklearn`'s Incremental PCA class 5 to reduce the dimensionality of the MNIST dataset down to 154 dimensions, just like before.

you must call the `partial_fit()` method with each mini-batch, rather than the `fit()` method with the whole training set.

C.R. 13 python
```python
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

Alternatively, you can use NumPy's `memmap` class[6], which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory; the class loads only the data it needs in memory, when it needs it.

[6] Memory-mapped files are used for accessing small segments of large files on disk, without reading the entire file into memory

To demonstrate this, let's first create a memory-mapped (memmap) file and copy the MNIST training set to it, then call `flush()` to ensure that any data still in the cache gets saved to disk. In real life, `X_train` would typically not fit in memory, so you would load it chunk by chunk and save each chunk to the right part of the memmap array:

C.R. 14 python
```python
filename = "my_mnist.mmap"
X_mmap = np.memmap(filename, dtype='float32', mode='write', shape=X_train.shape)
X_mmap[:] = X_train  # could be a loop instead, saving the data chunk by chunk
X_mmap.flush()
```

Next, we can load the memmap file and use it like a regular NumPy array. Let's use the `IncrementalPCA` class to reduce its dimensionality. Since this algorithm uses only a small part of the array at any given time, memory usage remains under control. This makes it possible to call the usual `fit()` method instead of `partial_fit()`, which is quite convenient:

C.R. 15 python
```python
X_mmap = np.memmap(filename, dtype="float32", mode="readonly").reshape(-1, 784)
batch_size = X_mmap.shape[0] // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mmap)
```

Only the raw binary data is saved to disk, so specify the data type and shape of the array when you load it. If you omit the shape, `np.memmap()` returns a 1D array.

For very high-dimensional datasets, PCA can be too slow. As we saw previously, even if you use randomized PCA its computational complexity is still $\mathcal{O}\left(m \times d^2\right) + \mathcal{O}\left(d^3\right)$, so the target number of dimensions d must not be too large. If you are dealing with a dataset with tens of thousands of features or more (e.g., images), then training may become much too slow: in this case, you should consider using random projection instead.

4.4 Random Projection

As its name suggests, the random projection algorithm projects the data to a lower-dimensional space using a random linear projection. This may sound counter intuitive, but turns out such a random projection is actually very likely to preserve distances fairly well, as was demonstrated mathematically by William B. Johnson and Joram Lindenstrauss in a famous lemma. So, two similar instances will remain similar after the projection, and two very different instances will remain very different.

Obviously, the more dimensions you drop, the more information is lost, and the more distances get distorted. So how can you choose the optimal number of dimensions? Well, Johnson and Lindenstrauss came up with an equation that determines the minimum number of dimensions to preserve in order to ensure—with high probability—that distances won't change by more than a given tolerance. For example, if you have a dataset containing m = 5,000 instances with n = 20,000 features each, and you don't want the squared distance between any two instances to change by more than ϵ = 10%,6 then you should project the data down to d dimensions, with d ≥ 4 log(m) / (1*2* ϵ2 - 1*3* ϵ3), which is 7,300 dimensions. That's quite a significant dimensionality reduction! Notice that the equation does not use n, it only relies on m and ϵ. This equation is implemented by the `johnson_lindenstrauss_min_dim()` function:
Now we can just generate a random matrix P of shape [d, n], where each item is sampled randomly from a Gaussian distribution with mean 0 and variance 1 / d, and use it to project a dataset from n dimensions down to d:
That's all there is to it! It's simple and efficient, and no training is required: the only thing the algorithm needs to create the random matrix is the dataset's shape. The data itself is not used at all. `sklearn` offers a GaussianRandomProjection class to do exactly what we just did: when you call its fit() method, it uses `johnson_lindenstrauss_min_dim()` to determine the output dimensionality, then it generates a random matrix, which it stores in the `components_` attribute. Then when you call transform(), it uses this matrix to perform the projection. When creating the transformer, you can set eps if you want to tweak ϵ (it defaults to 0.1), and `n_components` if you want to force a specific target dimensionality d. The following code example gives the same result as the preceding code (you can also verify that `gaussian_rnd_proj.components_` is equal to P):
`sklearn` also provides a second random projection transformer, known as SparseRandomProjection. It determines the target dimensionality in the same way, generates a random matrix of the same shape, and performs the projection identically. The main difference is that the random matrix is sparse. This means it uses much less memory: about 25 MB instead of almost 1.2 GB in the preceding example! And it's also much faster, both to generate the random matrix and to reduce dimensionality: about 50% faster in this case. Moreover, if the input is sparse, the transformation keeps it sparse (unless you set `dense_output=True`). Lastly, it enjoys the same distance- preserving property as the previous approach, and the quality of the dimensionality reduction is comparable. In short, it's usually preferable touse this transformer instead of the first one, especially for large or sparse datasets. The ratio r of nonzero items in the sparse random matrix is called its density. By default, it is equal to 1/n. With 20,000 features, this means that only 1 in 141 cells in the random matrix is nonzero: that's quite sparse! You can set the density hyperparameter to another value if you prefer. Each cell in the sparse random matrix has a probability r of being nonzero, and each nonzero value is either -v or +v (both equally likely), where v = 1/dr. If you want to perform the inverse transform, you first need to compute the pseudo-inverse of the components matrix using SciPy's pinv() function, then multiply the reduced data by the transpose of the pseudo-inverse:

Summary

> Random projection is a simple, fast, memory-efficient, and powerful dimensionality reduction algorithm that we should keep in mind, especially when dealing with high-dimensional datasets.

4.5 Locally Linear Embedding

Locallylinearembedding(LLE) is a non-linear dimensionality reduction (NLDR) technique. It is a manifold learning technique whici does **NOT** rely on projections, unlike PCA and random projection. In a nutshell, LLE works by first measuring how each training instance linearly relates to its nearest neighbors, and then looking for a low-dimensional representation of the training set where these local relationships are best preserved. This approach makes it good at unrolling twisted manifolds, especially when there is not too much noise. The following code makes a Swiss roll, then uses `sklearn`'s `LocallyLinearEmbedding` class to unroll it:
The variable `t` is a 1D NumPy array containing the position of each instance along the rolled axis of the Swiss roll. We don't use it in this example, but it can be used as a target for a nonlinear regression task. The resulting 2D dataset is shown in Figure 8-10. As you can see, the Swiss roll is completely unrolled, and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the unrolled Swiss roll should be a rectangle, not this kind of stretched and twisted band. Nevertheless, LLE did a pretty good job of modeling the manifold.

Operation Principle

For each training instance $\mathbf{x}_{(i)}$, the algorithm identifies its k-nearest neighbours, then tries to reconstruct $\mathbf{x}_{(i)}$ as a linear function of these neighbors. More specifically, it tries to find the weights $w_{i,j}$ such that the squared distance between $\mathbf{x}_{(i)}$ and:

$$\sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(1)}$$

is as **small as possible**, assuming $w_{i,j} = 0$ if $\mathbf{x}^{(i)}$ is not one of the k-nearest neighbors of $\mathbf{x}_{(i)}$. Thus the first step of LLE is the constrained optimization problem:

$$\mathbf{W}\,[h] =$$

, where W is the weight matrix containing all the weights wi,j. The second constraint simply normalizes the weights for each training instance x(i).
After this step, the weight matrix W(containing the weights w i,j) encodes the local linear relationships between the training instances. The second step is to map the training instances into a d-dimensional space (where d < n) while preserving these local relationships as much as possible. If z(i) is the image of x(i) in this d-dimensional space, then we want the squared distance between z(i) and ∑ j=1 m w i,j z (j) to be as small as possible. This idea leads to the unconstrained optimization problem described in Equation 8-5. It looks very similar to the first step, but instead of keeping the instances fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances' images in the low-dimensional space. Note that Z is the matrix containing all z(i).
`sklearn`'s LLE implementation has the following computational complexity: O(m log(m)n log(k)) for finding the k-nearest neighbors, O(mnk3) for optimizing the weights, and O(dm2) for constructing

the low-dimensional representations. Unfortunately, the m2 in the last term makes this algorithm scale poorly to very large datasets. As you can see, LLE is quite different from the projection techniques, and it's significantly more complex, but it can also construct much better low- dimensional representations, especially if the data is nonlinear.

Model Speed of Dimensionality Reduction

LoadtheMNISTdataset(introducedinChapter3)andsplititintoa training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a random forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%. Train a new random forest classifier on the reduced dataset and see how long it takes. Was training much faster? Next, evaluate the classifier on the test set. How does it compare to the previous classifier? Try again with an SGDClassifier. How much does PCA help now?
The MNIST dataset was loaded earlier.

C.R. 16 python
```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False, parser="auto")

X_train = mnist.data[:60000]
y_train = mnist.target[:60000]

X_test = mnist.data[60000:]
y_test = mnist.target[60000:]
```

Train a Random Forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set.

C.R. 17 python
```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
```

C.R. 18 python
```
import time
start_time = time.time()
rnd_clf.fit(X_train, y_train)
print("--- %s seconds ---" % (time.time() - start_time))
```

C.R. 19 python
```
from sklearn.metrics import accuracy_score

y_pred = rnd_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95

C.R. 20 python
```
from sklearn.decomposition import PCA

pca = PCA(n_components=0.95)
X_train_reduced = pca.fit_transform(X_train)
```

Train a new Random Forest classifier on the reduced dataset and see how long it takes. Was training much faster?

C.R. 21 python
```
rnd_clf_with_pca = RandomForestClassifier(n_estimators=100, random_state=42)

start_time = time.time()
```

C.R. 22 python
```
rnd_clf_with_pca.fit(X_train_reduced, y_train)
print("--- %s seconds ---" % (time.time() - start_time))
```

Oh no! Training is actually about twice slower now! How can that be? Well, as we saw in this chapter, dimensionality reduction does not always lead to faster training time: it depends on the dataset, the model and the training algorithm. See figure 8-6 (the `manifold_decision_boundary_plot*` plots above). If you try SGDClassifier instead of RandomForestClassifier, you will find that training time is reduced by a factor of 3 when using PCA. Actually, we will do this in a second, but first let's check the precision of the new random forest classifier.
Exercise: Next evaluate the classifier on the test set: how does it compare to the previous classifier?

C.R. 23 python
```
from sklearn.linear_model import SGDClassifier

start_time = time.time()
sgd_clf = SGDClassifier(random_state=42)
print("--- %s seconds ---" % (time.time() - start_time))
```

It is common for performance to drop slightly when reducing dimensionality, because we do lose some potentially useful signal in the process. However, the performance drop is rather severe in this case. So PCA really did not help: it slowed down training and reduced performance.
Exercise: Try again with an SGDClassifier. How much does PCA help now?

Visualising Data - I — 1 Example

Use t-SNE to reduce the first 5,000 images of the MNIST dataset down to two dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to represent each image's target class.

Solution

Let's limit ourselves to the first 5,000 images of the MNIST training set, to speed things up a lot.

C.R. 24 python
```
X_sample, y_sample = X_train[:5000], y_train[:5000]
```

Let's use t-SNE to reduce dimensionality down to 2D so we can plot the dataset:

C.R. 25 python
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, init="random", learning_rate="auto",
            random_state=42)

X_reduced = tsne.fit_transform(X_sample)
```

Now let's use Matplotlib's scatter() function to plot a scatterplot, using a different color for each digit:

C.R. 26 python
```
plt.figure(figsize=(13, 10))
plt.scatter(X_reduced[:, 0], X_reduced[:, 1],
            c=y_sample.astype(np.int8), cmap="jet", alpha=0.5)
plt.axis('off')
plt.colorbar()
cp.store_fig("t-sne-plot", close=True)
```

Most digits are nicely separated from the others, even though t-SNE wasn't given the targets: it just identified clusters of similar images. But there is still a bit of overlap. For example, the 3s and the 5s overlap a lot (on the right side of the plot), and so do the 4s and the 9s (in the top-right corner).

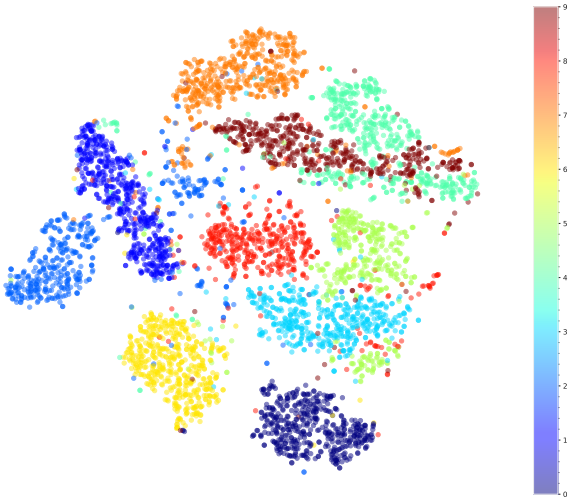

Figure 4.8

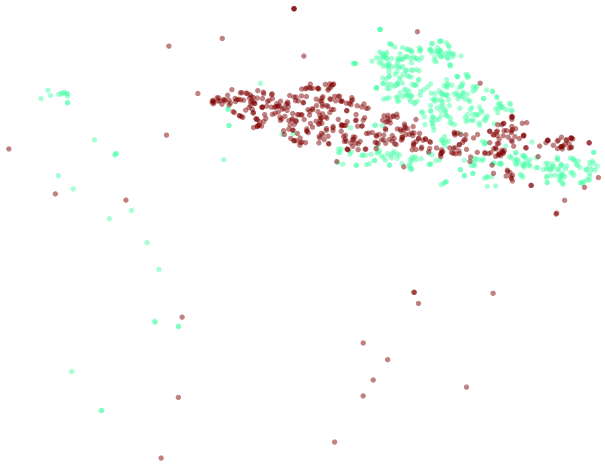

Figure 4.9

Let's focus on just the digits 4 and 9:
Let's see if we can produce a nicer image by running t-SNE on just these 2 digits:

C.R. 27 python

```
idx = (y_sample == '4') | (y_sample == '9')
X_subset = X_sample[idx]
y_subset = y_sample[idx]

tsne_subset = TSNE(n_components=2, init="random", learning_rate="auto",
                   random_state=42)
X_subset_reduced = tsne_subset.fit_transform(X_subset)
```

C.R. 28 python

```
plt.figure(figsize=(13, 10))
for digit in ('4', '9'):
    plt.scatter(X_subset_reduced[y_subset == digit, 0],
                X_subset_reduced[y_subset == digit, 1],
```

C.R. 29
python
```
                c=[cmap(float(digit) / 9)], alpha=0.5)
plt.axis('off')
cp.store_fig("fn-sne-plot-nice", close=True)
```

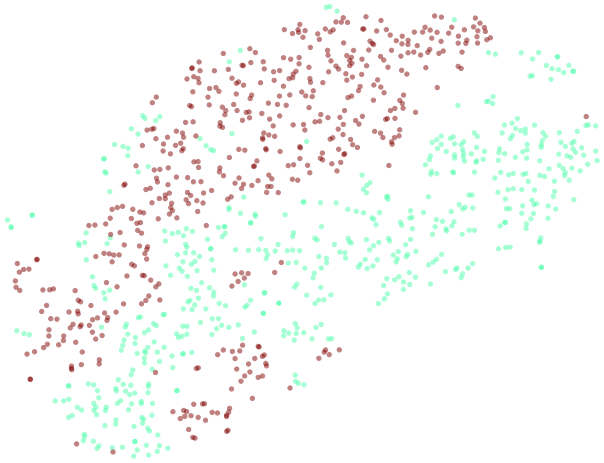

Figure 4.10

That's much better, although there's still a bit of overlap. Perhaps some 4s really do look like 9s, and vice versa. It would be nice if we could visualize a few digits from each region of this plot, to understand what's going on. In fact, let's do that now.

Visualising Data - II 2 **Example**

Alternatively, you can replace each dot in the scatterplot with the corresponding instance's class (a digit from 0 to 9), or even plot scaled-down versions of the digit images themselves (if you plot all digits, the visualization will be too cluttered, so you should either draw a random sample or plot an instance only if no other instance has already been plotted at a close distance). You should get a nice visualization with well-separated clusters of digits.

Solution

Let's create a `plot_digits()` function that will draw a scatterplot (similar to the above scatterplots) plus write colored digits, with a minimum distance guaranteed between these digits. If the digit images are provided, they are plotted instead.

C.R. 30
python
```
from sklearn.preprocessing import MinMaxScaler
from matplotlib.offsetbox import AnnotationBbox, OffsetImage

def plot_digits(X, y, min_distance=0.04, images=None, figsize=(13, 10)):
    # Let's scale the input features so that they range from 0 to 1
    X_normalized = MinMaxScaler().fit_transform(X)
    # Now we create the list of coordinates of the digits plotted so far.
    # We pretend that one is already plotted far away at the start, to
    # avoid `if` statements in the loop below
    neighbors = np.array([[10., 10.]])
    # The rest should be self-explanatory
    plt.figure(figsize=figsize)
    cmap = plt.cm.jet
    digits = np.unique(y)
    for digit in digits:
```

C.R. 31
python
```
        plt.scatter(X_normalized[y == digit, 0], X_normalized[y == digit, 1],
                    c=[cmap(float(digit) / 9)], alpha=0.5)
    plt.axis("off")
    ax = plt.gca()  # get current axes
    for index, image_coord in enumerate(X_normalized):
        closest_distance = np.linalg.norm(neighbors - image_coord, axis=1).min()
        if closest_distance > min_distance:
            neighbors = np.r_[neighbors, [image_coord]]
            if images is None:
                plt.text(image_coord[0], image_coord[1], str(int(y[index])),
                         color=cmap(float(y[index]) / 9),
                         fontdict={"weight": "bold", "size": 16})
            else:
                image = images[index].reshape(28, 28)
                imagebox = AnnotationBbox(OffsetImage(image, cmap="binary"),
                                          image_coord)
                ax.add_artist(imagebox)
```

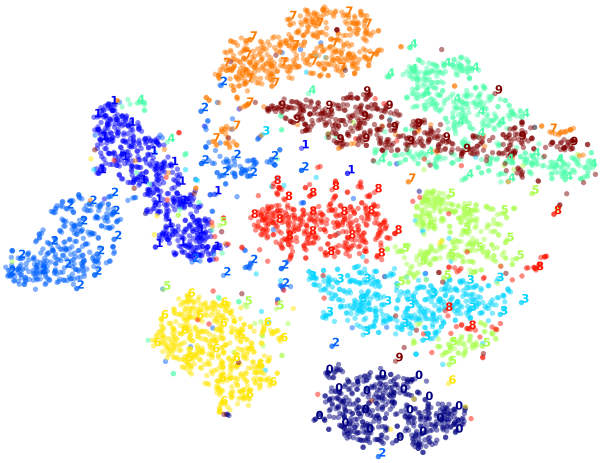

Figure 4.11

Well that's okay, but not that beautiful. Let's try with the digit images:

Chapter 5

Unsupervised Learning

Table of Contents

While most ML application nowadays are based on supervised learning[1], the vast majority of the available data is unlabeled:

> This means we have the input features (X), but do not have the labels (y). For example we can have the login information of users to a website but have no idea of their name, sex, occupation, etc..

[1] This is where most companies spend most of their moneys on

There is a good quote by computer scientist *Yann LeCun* (Former Facebook AI Chief) given in NIPS[2] 2016 which gives a good idea on the types of learning in ML [14]:

[2] Neural Information Processing Systems

> *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.*

In other words, there is a huge potential in unsupervised learning that we have only barely started to sink our teeth into.

Let's image a scenario. Let's say we want to create a system which will take a few pictures of each item on a manufacturing production line and detect which items are defective. We can easily create a system which will take pictures automatically, and this might give you thousands of pictures every day. We can then build a reasonably large dataset in just a few weeks. However we will hit a road block as there are **no labels**.

To train a regular binary classifier to predict whether an item is defective or not, will need to label every single picture either as defective or normal. This will generally require human experts to sit down and manually go through all the pictures. As you can imagine, this is rather a long, costly, and tedious task, so it will usually only be done on a small subset of the available pictures. This in turn will make the labelled dataset quite small, and the classifier's performance will be less than optimal.

> In addition, every time the company makes any change to its products, the whole process will need to be started over from scratch.

These restrictions can make ML either a tedious task at best or a massive time sink at worst. Wouldn't it be great if the algorithm could just exploit the unlabelled data without needing humans to label every picture?

This is where unsupervised learning shows its performance..

In Chapter 8 we looked at the most common unsupervised learning task, dimensionality reduction and in this chapter we will look at a few more unsupervised tasks:

Clustering The goal is to group similar instances together into clusters. Clustering is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and much more.

[3] This is also known as outlier detection.

Anomaly Detection [3] The goal is to learn what "normal" data looks like, and then use that to detect abnormal instances. These instances are called anomalies, or outliers, while the normal instances are called inliers. Anomaly detection is useful in a wide variety of applications, such as fraud detection, detecting defective products in manufacturing, identifying new trends in time series, or removing outliers from a dataset before training another model, which can significantly improve the performance of the resulting model.

Density Estimation This is the task of estimating the probability density function (PDF) of the random process that generated the dataset. Density estimation is commonly used for anomaly detection: instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualisation.

5.1 Clustering Algorithms

As we are enjoying our hike through the mountains of Tirol, we stumble upon a plant we have never seen before. It could be edelweiss or maybe something else. We can't tell. We look around and we notice a few more. They are not identical, yet they are sufficiently similar for we to know that they most likely belong to the same species. We may need a botanist, or a local, to tell you what species that is, but we certainly don't need an expert to identify groups of similar-looking objects.

This is called clustering:

> It is the task of identifying similar instances and assigning them to clusters, or groups of similar instances without knowing what the instance really is.

Similar to classification, each instance gets assigned to a **group**. However, unlike classification, clustering is an unsupervised task. Consider **Fig.** 5.1 : on the left is the iris dataset, which we worked

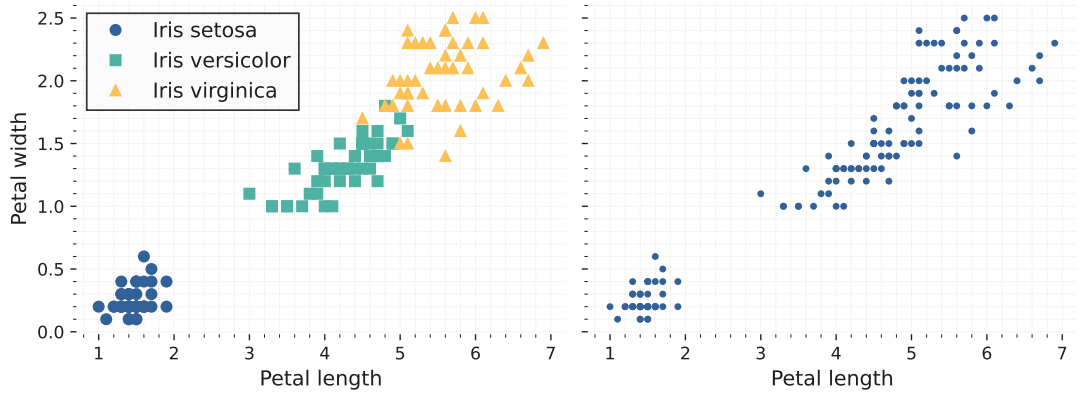

Figure 5.1: lassification (left) versus clustering (right).

before, where each instance's species[4] is represented with a different marker. It is a labelled dataset, for which classification algorithms such as logistic regression, SVMs, or random forest classifiers are well suited.

[4] i.e., its class

On the right is the same dataset, but without the labels, so we cannot use a classification algorithm anymore. This is where clustering algorithms step in: many of them can easily detect the lower-left cluster. It is also quite easy to see with our own eyes, but it is not so obvious that the upper-right cluster is composed of two (2) distinct subclusters. That said, the dataset has two additional features (*sepal length* and *width*) that are not represented here, and clustering algorithms can make good use of all features, so in fact they identify the three clusters fairly well [5]

[5] e.g., using a Gaussian mixture model, only 5 instances out of 150 are assigned to the wrong cluster.

Clustering is used in a wide variety of applications, including:

Customer Segmentation We can cluster our customers based on their purchases and their activity on our website. This is useful to understand who our customers are and what they need, so we can adapt our products and marketing campaigns to each segment [8].

> Customer segmentation can be useful in recommender systems to suggest content that other users in the same cluster enjoyed.

Data analysis When we analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.

Dimensionality reduction Once a dataset has been clustered, it is usually possible to measure each instance's affinity with each cluster. Here, affinity is any measure of how well an instance fits into a cluster. Each instance's feature vector $\mathbf{x}$ can then be replaced with the vector of its cluster affinities. If there are k clusters, then this vector is k-dimensional.

> The new vector is typically much lower-dimensional than the original feature vector, but it can preserve enough information for further processing.

Feature engineering The cluster affinities can often be useful as extra features. For example, we used k-means before to add geographic cluster affinity features to the California housing dataset, and they helped us get better performance.

Anomaly detection Any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if we have clustered the users of our website based on their behavior, we can detect users with unusual behavior, such as an unusual number of requests per second [9].

Semi-supervised learning If we only have a few labels, we could perform clustering and propagate the labels to all the instances in the same cluster. This technique can greatly increase the number of labels available for a subsequent supervised learning algorithm, and thus improve its performance [2].

Search engines Some search engines let you search for images that are similar to a reference image. To build such a system, we first apply a clustering algorithm to all the images in our database. This allows similar images to end up in the same cluster. Then when a user provides a reference image, all we'd need to do is use the trained clustering model to find this image's cluster, and we could then simply return all the images from this cluster [13].

Image segmentation By clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to considerably reduce the number of different colors in an image. Image segmentation is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object [4].

There is **no universal definition of what a cluster is** as it really depends on the context, and different algorithms will capture different kinds of clusters. Some algorithms, for example, look for instances centered around a particular point, called a **centroid**. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters. And the list goes on.

In this section of our chapter, we will look at two (2) popular clustering algorithms, k-means and DBSCAN, and explore some of their applications, such as nonlinear dimensionality reduction, semi-supervised learning, and anomaly detection.

5.1.1 k-means

Consider the unlabelled dataset represented in **Fig.** 5.2 . It is clear to us to say we can clearly see five (5) blobs of instances. The *k*-means algorithm is a simple algorithm capable of clustering this kind of dataset very quickly and efficiently, often in just a few iterations. It was proposed by *Stuart Lloyd* at Bell Labs in 1957 as a technique for Pulse Code Modulation (PCM), but it was only published outside of the company in 1982 [10]. In 1965, *Edward W. Forgy* had published virtually the same algorithm [6], so *k*-means is sometimes referred to as the Lloyd-Forgy algorithm.
Let's train a *k*-means clusterer on this dataset. It will try to find each blob's center and assign each instance to the closest blob:

C.R.1 python

```python
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

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

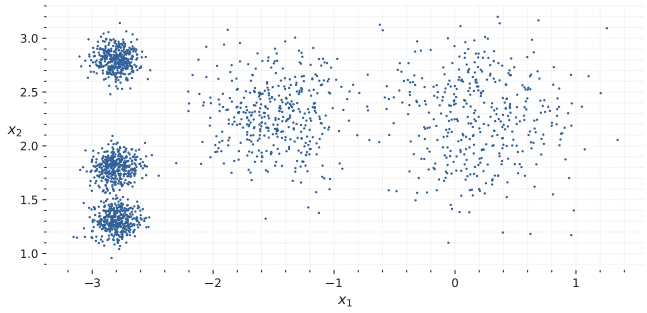

Figure 5.2: An unlabelled dataset composed of five blobs of instances.

C.R. 2
python
```
k = 5
kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
y_pred = kmeans.fit_predict(X)
```

> We have to specify the number of clusters k that the algorithm must find

In this example, as said previously, it is obvious k should be set to five (5), but in general it is not that easy. Each instance will be assigned to one of the five (5) clusters. In the context of clustering, an instance's label is the index of the cluster to which the algorithm assigns this instance. This should not to be confused with the class labels in classification, which are used as targets[6]. The `KMeans` instance preserves the predicted labels of the instances it was trained on, available via the `labels_` instance variable:

[6] remember, clustering is an unsupervised learning task

C.R. 3
python
```
print(y_pred)
```

text
```
[0 0 4 ... 3 1 0]
```

C.R. 4
python
```
y_pred is kmeans.labels_
```

text
```
True
```

We can also take a look at the five (5) centroids that the algorithm found:

C.R. 5
python
```
print(kmeans.cluster_centers_)
```

text
```
[[-2.80214068  1.55162671]
 [ 0.08703534  2.58438091]
 [-1.46869323  2.28214236]
 [-2.79290307  2.79641063]
 [ 0.31332823  1.96822352]]
```

We can easily assign new instances to the cluster whose centroid is closest:

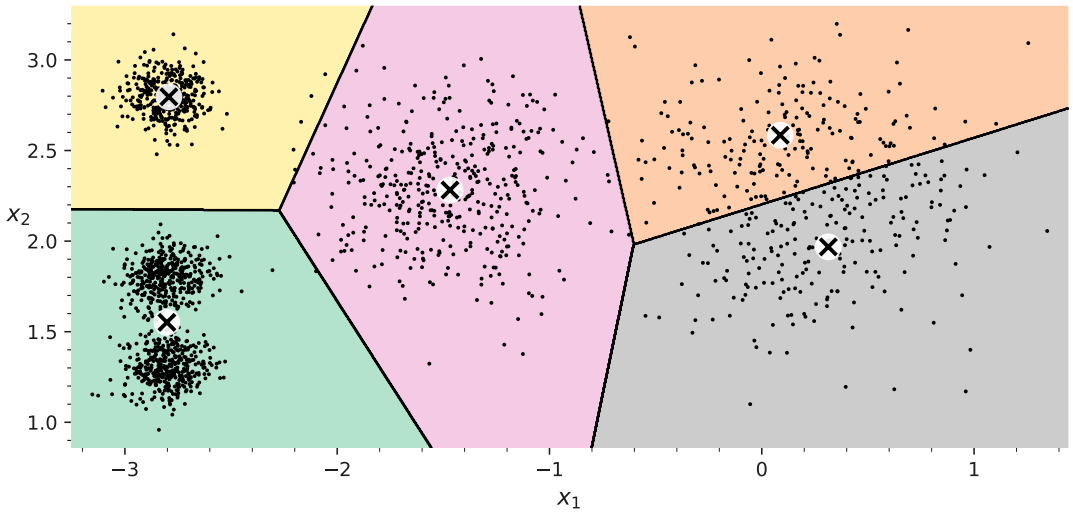

Figure 5.3: k-means decision boundaries (Voronoi tessellation)

C.R. 6
```python
import numpy as np

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

```text
[4 4 3 3]
```

[7] a type of tessellation pattern in which a number of points scattered on a plane subdivides in exactly n cells enclosing a portion of the plane that is closest to each point.

If we plot the cluster's decision boundaries, we get a Voronoi tessellation[7] which can be seen in **Fig. 5.3** , where each centroid is represented with an X.

The vast majority of the instances were clearly assigned to the appropriate cluster, but a good part of the instances were mislabelled, Such as parts in the lower left where there is obviously two centre points, but the algorithm decided there is only one. Indeed, the k-means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the **centroid**.

[8] This is a similar behaviour to hard v. soft voting we encountered in Random Forest

Instead of assigning each instance to a single cluster, which is called **hard clustering**, it can be useful to give each instance a score per cluster, which is called **soft clustering** [8] The score can be the distance between the instance and the centroid or a similarity score, such as the Gaussian Radial Basis Function (RBF). In the `KMeans` class, the `transform()` method measures the distance from each instance to every centroid:

C.R. 7
```python
print(kmeans.transform(X_new).round(2))
```

```text
[[2.84 0.59 1.5  2.9  0.31]
 [5.82 2.97 4.48 5.85 2.69]
 [1.46 3.11 1.69 0.29 3.47]
 [0.97 3.09 1.55 0.36 3.36]]
```

In the aforementioned example, the first instance in `X_new` is located at a distance of about 2.84 from the 1st centroid, 0.59 from the 2nd centroid, 1.5 from the 3rd centroid, 2.9 from the 4th centroid, and 0.31 from the 5th centroid.

If we have a high-dimensional dataset and we transform it this way, we end up with a k-dimensional dataset: this transformation can be a very efficient nonlinear dimensionality reduction technique. Alternatively, we can use these distances as extra features to train another model.

The Operation Principle

Let's try to understand k-means via an example. Suppose we were given the centroids. We could easily label all the instances in the dataset by assigning each of them to the cluster whose centroid is closest. Or, if we were given all the instance labels, we could easily locate each cluster's centroid by computing the mean of the instances in that cluster.

But we are given neither the labels nor the centroids, so how can we proceed?

We start by placing the centroids randomly (e.g., by picking k instances at random from the dataset and using their locations as centroids). Then label the instances, update the centroids, label the instances, update the centroids, and so on until the centroids stop moving.

> The algorithm is guaranteed to converge in a finite number of steps.

That's because the mean squared distance between the instances and their closest centroids can only go down at each step, and since it cannot be negative, it's guaranteed to converge. We can see the algorithm in action in **Fig.** 5.4 :

Let's try to explain the behaviour of the figure.

- the centroids are initialised randomly (top left)
- then the instances are labelled (top right),
- the centroids are updated (center left),
- the instances are relabelled (center right),

and so on. As we can see, in just three (3) iterations the algorithm has reached a clustering that seems close to optimal.

Computational Complexity

The computational complexity of the algorithm is generally linear with regard to the number of instances m, the number of clusters k, and the number of dimensions n. However, this is only true when the data has a clustering structure. If it does not, then in the worst-case scenario the complexity can increase exponentially with the number of instances.

In practice, this rarely happens, and k-means is generally one of the fastest clustering algorithms.

Although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): whether it does or not depends on the centroid initialisation. **Fig.**

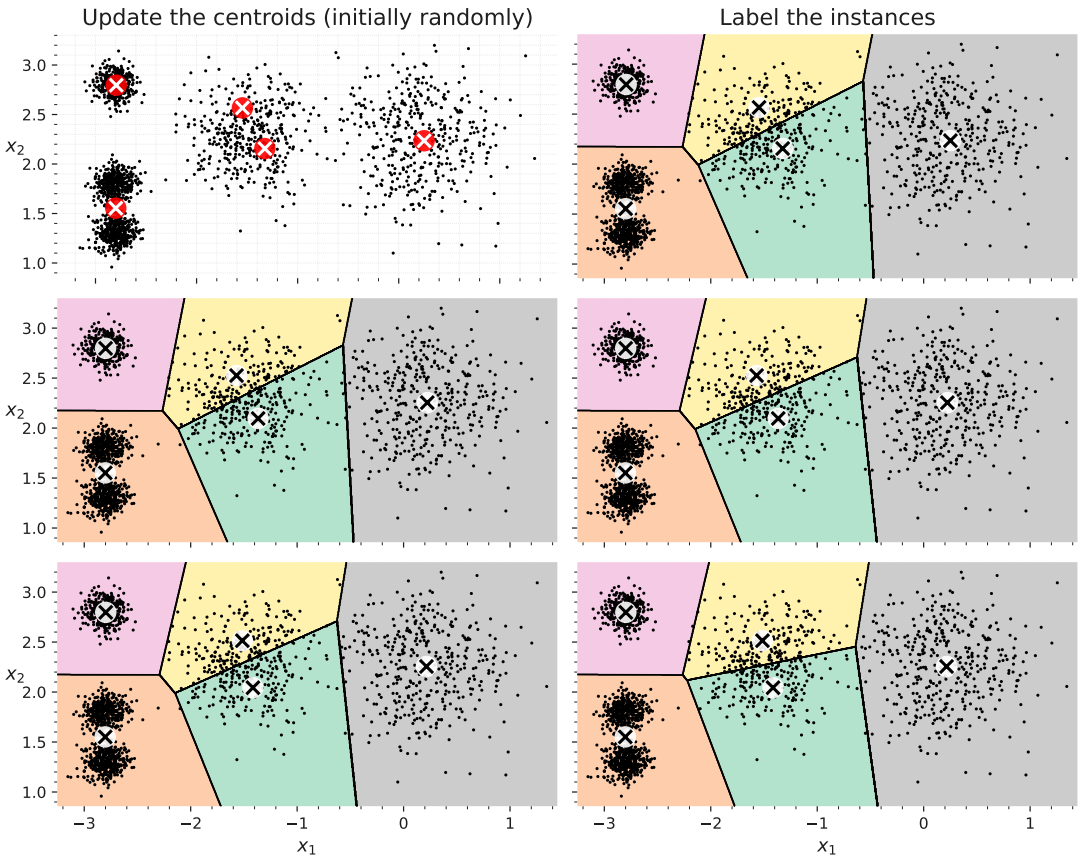

Figure 5.4: The k-means algorithm.

5.5 shows two (2) sub-optimal solutions that the algorithm can converge to if we are not lucky with the random initialisation step.
Let's take a look at a few ways we can mitigate this risk by improving the centroid initialisation.

Centroid initialisation methods

If we happen to know approximately where the centroids should be (i.e., if we ran another clustering algorithm earlier), then we can set the init hyperparameter to a numpy array containing the list of centroids, and set `n_init` to 1:

C.R. 8 python
```
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,
                random_state=42)
kmeans.fit(X)
```

Another solution is to run the algorithm multiple times with different random initialisations and keep the best solution. The number of random initialisation is controlled by the `n_init` hyperparameter:

> by default it is equal to 10, which means that the whole algorithm described earlier runs 10 times when we call `fit()`, and `sklearn` keeps the best solution.

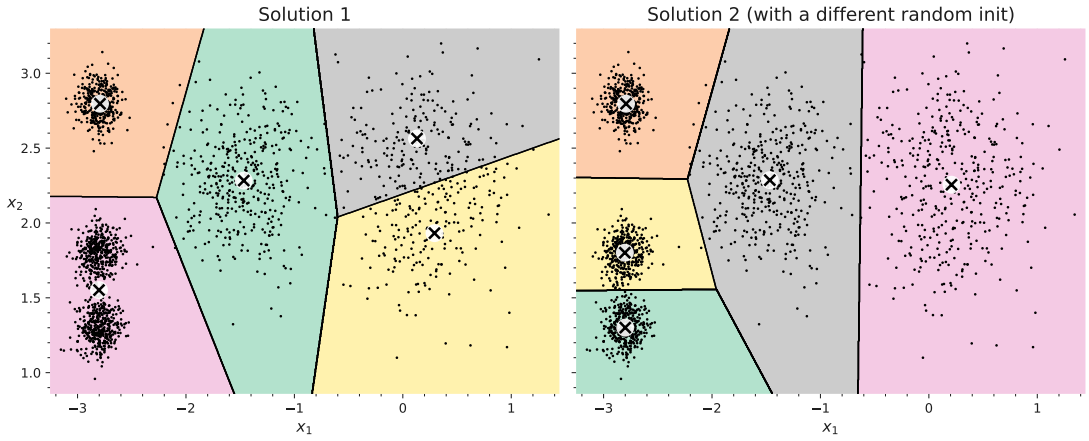

Figure 5.5: Suboptimal solutions due to unlucky centroid initialisation.

But how exactly does it know which solution is the best? It uses a performance metric. That metric is called the **model's inertia**, which is the sum of the squared distances between the instances and their closest centroids.

It is roughly equal to 219.4 for the model on the left in **Fig.** 5.11 , 258.6 for the model on the right in **Fig.** 5.11 , and only 211.6 for the model in **Fig.** 5.3 . The `KMeans` class runs the algorithm `n_init` times and keeps the model with the lowest inertia.

In this example, the model in **Fig.** 5.3 will be selected (unless we are very unlucky with `n_init` consecutive random initialisation). For the curious, a model's inertia is accessible via the `inertia_` instance variable:

C.R. 9
```python
kmeans.inertia_
```

```text
211.59853725816836
```

The `score()` method returns the negative inertia (it's negative because a predictor's score() method must always respect `sklearn`'s "greater is better" rule: if a predictor is better than another, its `score()` method should return a greater score):

C.R. 10
```python
kmeans.score(X)
```

```text
-211.5985372581684
```

An important improvement to the k-means algorithm, k-means++, was proposed in a 2006 paper by David Arthur and Sergei Vassilvitskii [1]. They introduced a smarter initialisation step that tends to select centroids that are distant from one another, and this improvement makes the k-means algorithm much less likely to converge to a sub-optimal solution.

The paper showed, the additional computation required for the smarter initialisation step is well worth

it because it makes it possible to drastically reduce the number of times the algorithm needs to be run to find the optimal solution.

The k-means++ initialisation works as follows:

1. Take one centroid $\mathbf{c}^{(1)}$, chosen uniformly at random from the dataset,
2. Take the new centroid $\mathbf{c}^{(i)}$, choosing an instance $\mathbf{x}^{(i)}$ with probability:

$$p\left(\mathbf{x}^{(i)}\right) = \frac{D\left(\mathbf{x}^{(i)}\right)^2}{\sum_{j=1}^{m} D\left(\mathbf{x}^{(i)}\right)^2}$$

where $D\left(\mathbf{x}^{(i)}\right)^2$ is the distance between the instance $\mathbf{x}^{(i)}$ and the closest centroid that was already chosen.

> This probability distribution ensures that instances farther away from already chosen centroids are much more likely to be selected as centroids.

3. Repeat the previous step until all k centroids have been chosen.

The `KMeans` class uses this initialisation method by default. To force it to use the original method (i.e., picking k instances randomly to define the initial centroids), then we can set the init hyperparameter to `"random"`.

> We will rarely need to do this.

Accelerated and mini-batch

Another improvement to the k-means algorithm was proposed in a 2003 paper by *Charles Elkan* [5]. On some large datasets with many clusters, the algorithm can be accelerated by avoiding many unnecessary distance calculations. Elkan achieved this by exploiting the triangle inequality[9] and by keeping track of lower and upper bounds for distances between instances and centroids. However, Elkan's algorithm does not always accelerate training, and sometimes it can even slow down training significantly as it depends on the dataset.

[9] The triangle inequality is $AC \leq AB + BC$, where A, B and C are three points and AB, AC, and BC are the distances between these points.

> To give it a try, set `algorithm="elkan"`.

Yet another important variant of the k-means algorithm was proposed in a 2010 paper by David Sculley [12]. Instead of using the full dataset at each iteration, the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration. This speeds up the algorithm[10] and makes it possible to cluster huge datasets that do not fit in memory. `sklearn` implements this algorithm in the `MiniBatchKMeans` class, which we can use just like the `KMeans` class:

[10] typically by a factor of three to four

C.R. 11 python

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=10, batch_size=10,
↪  n_init=3, random_state=42)
minibatch_kmeans.fit(X_memmap)
```

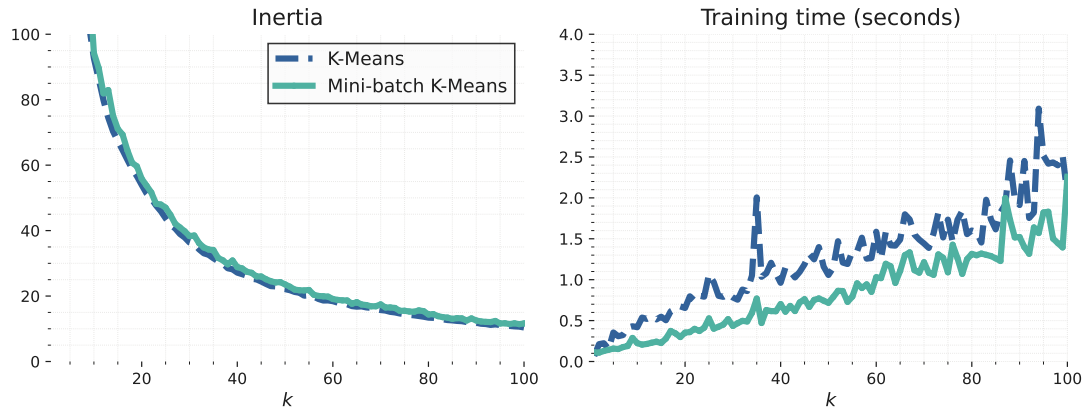

Figure 5.6: Mini-batch k-means has a higher inertia than k-means (left) but it is much faster (right), especially as k increases.

If the dataset does not fit in memory, the simplest option is to use the `memmap` class. Alternatively, we can pass one mini-batch at a time to the `partial_fit()` method, but this will require much more work, as we will need to perform multiple initialisations and select the best one ourselves.

Although the mini-batch k-means algorithm is much faster than the regular k-means algorithm, its inertia is generally slightly worse. We can see this in **Fig.** 5.6 : the plot on the left compares the inertiae of mini-batch k-means and regular k-means models trained on the previous five-blobs dataset using various numbers of clusters k. The difference between the two curves is small, but visible. In the plot on the right, we can see that mini-batch k-means is roughly 1.5 - 2 times faster than regular k-means on this dataset.

Finding the optimal number of clusters

So far, we've set the number of clusters k to five (5) as it was obvious by looking at the data that this was the correct number of clusters. But in general, it won't be so easy to know how to set k, and the result might be quite bad if we set it to the wrong value. As we can see in **Fig.** 5.7 , for this dataset setting k to 3 or 8 results in fairly bad models.

We might be thinking that we could just pick the model with the lowest inertia. Unfortunately, it is not that simple. The inertia for $k = 3$ is about 653.2, which is much higher than for $k = 5$ with a value of 211.6. But with $k = 8$, the inertia is just 119.1.

> The inertia is not a good performance metric when trying to choose k as it keeps getting lower as we increase k.

Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. To see this visually Let's plot the inertia as a function of k. When we do this, the curve often contains an inflexion point called the elbow (see **Fig.** 5.8).
As we can see, the inertia drops very quickly as we increase k up to 4, but then it decreases much more slowly as we keep increasing k. This curve has roughly the shape of an arm, and there is an elbow at k = 4. So, if we did not know better, we might think 4 was a good choice as any lower value would be dramatic, while any higher value would not help much, and we might just be splitting perfectly good

clusters in half for no good reason.

This technique for choosing the best value for the number of clusters is rather coarse. A more precise[11] approach is to use the silhouette score, which is the mean silhouette coefficient over all the instances. An instance's silhouette coefficient is equal to:

[11] but also more computationally expensive

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

where a is the mean distance to the other instances in the same cluster (i.e., the mean intra-cluster distance) and b is the mean nearest-cluster distance[12]. The silhouette coefficient can vary between -1 and +1. A coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary and finally, a coefficient close to -1 means that the instance may have been assigned to the wrong cluster.

[12] the mean distance to the instances of the next closest cluster, defined as the one that minimizes b, excluding the instance's own cluster

To compute the silhouette score, we can use `sklearn`'s `silhouette_score()` function, giving it all the instances in the dataset and the labels they were assigned:

C.R. 12

```python
from sklearn.metrics import silhouette_score

silhouette_score(X, kmeans.labels_)
```

```text
0.655517642572828
```

Let's compare the silhouette scores for different numbers of clusters (see **Fig.** 5.9).
As we can see, this visualisation gives more information compared to the previous one:

> although it confirms that $k = 4$ is a very good choice, it also highlights the fact that $k = 5$ is quite good as well, at least much better than $k = 6$ or 7.

> This was not visible when comparing inertiae.

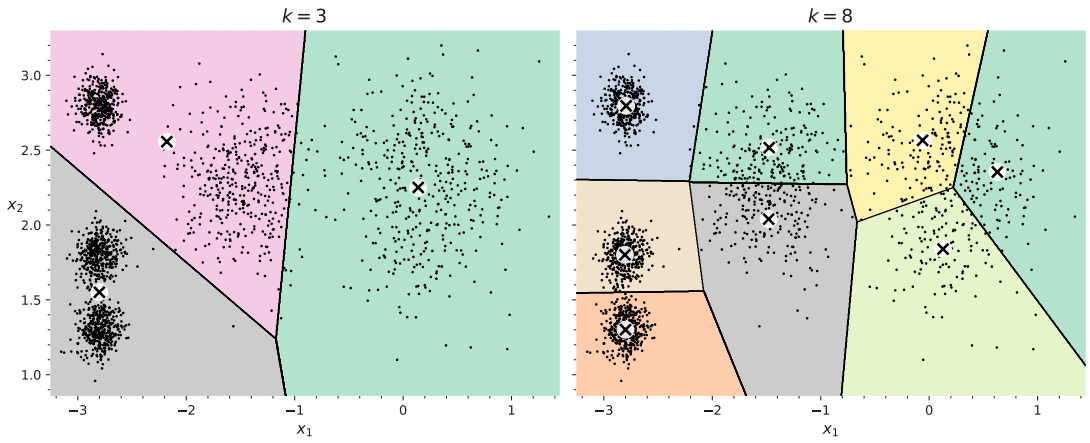

Figure 5.7: Bad choices for the number of clusters: when k is too small, separate clusters get merged (left), and when k is too large, some clusters get chopped into multiple pieces (right)

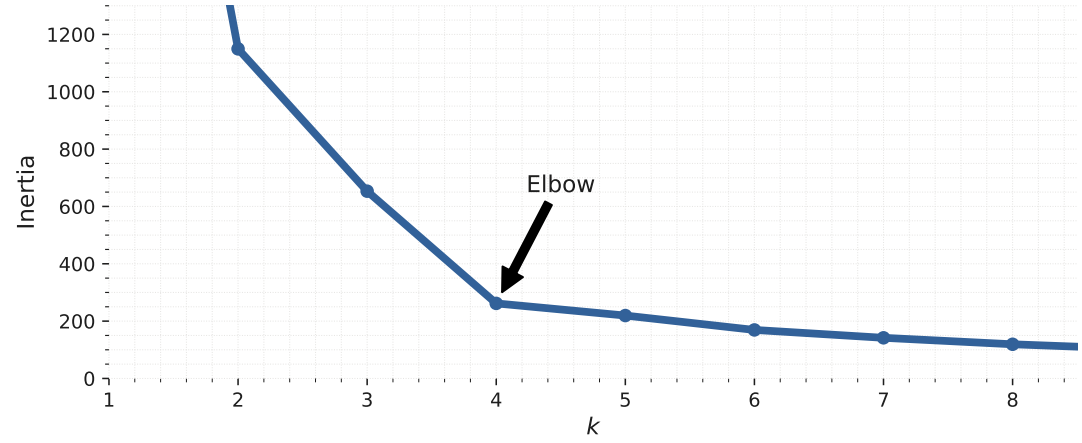

Figure 5.8: Plotting the inertia as a function of the number of clusters k

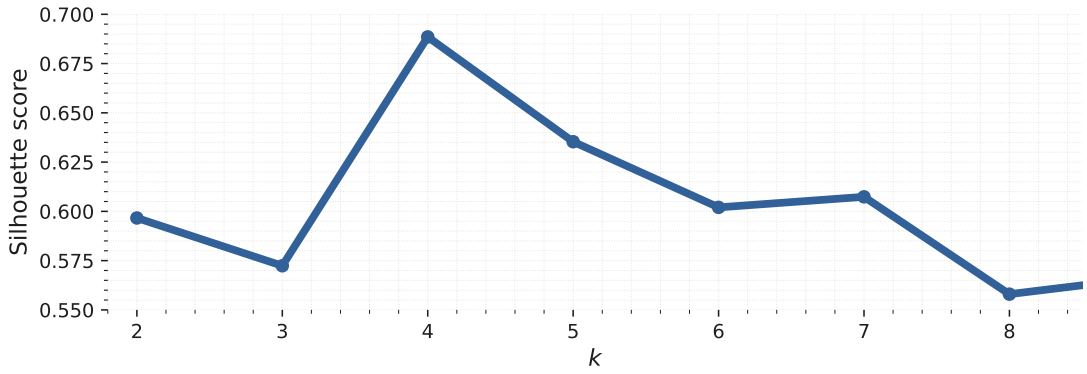

Figure 5.9: Selecting the number of clusters k using the silhouette score.

An even more informative visualisation is obtained when we plot every instance's silhouette coefficient, sorted by the clusters they are assigned to and by the value of the coefficient. This is called a silhouette diagram (see **Fig.** 5.10). Each diagram contains one knife shape per cluster. The shape's height indicates the number of instances in the cluster, and its width represents the sorted silhouette coefficients of the instances in the cluster[13].

[13] wider is better.

The vertical dashed lines represent the mean silhouette score for each number of clusters. When most instances in a cluster have a lower coefficient than this score (i.e., if many of the instances stop short of the dashed line, ending to the left of it), then the cluster is rather bad since this means its instances are much too close to other clusters. Here we can see that when k = 3 or 6, we get bad clusters. But when k = 4 or 5, the clusters look pretty good: most instances extend beyond the dashed line, to the right and closer to 1.0.

When k = 4, the cluster at index 2 (the second from the top) is rather big. When k = 5, all clusters have similar sizes. So, even though the overall silhouette score from k = 4 is slightly greater than for k = 5, it seems like a good idea to use k = 5 to get clusters of similar sizes.

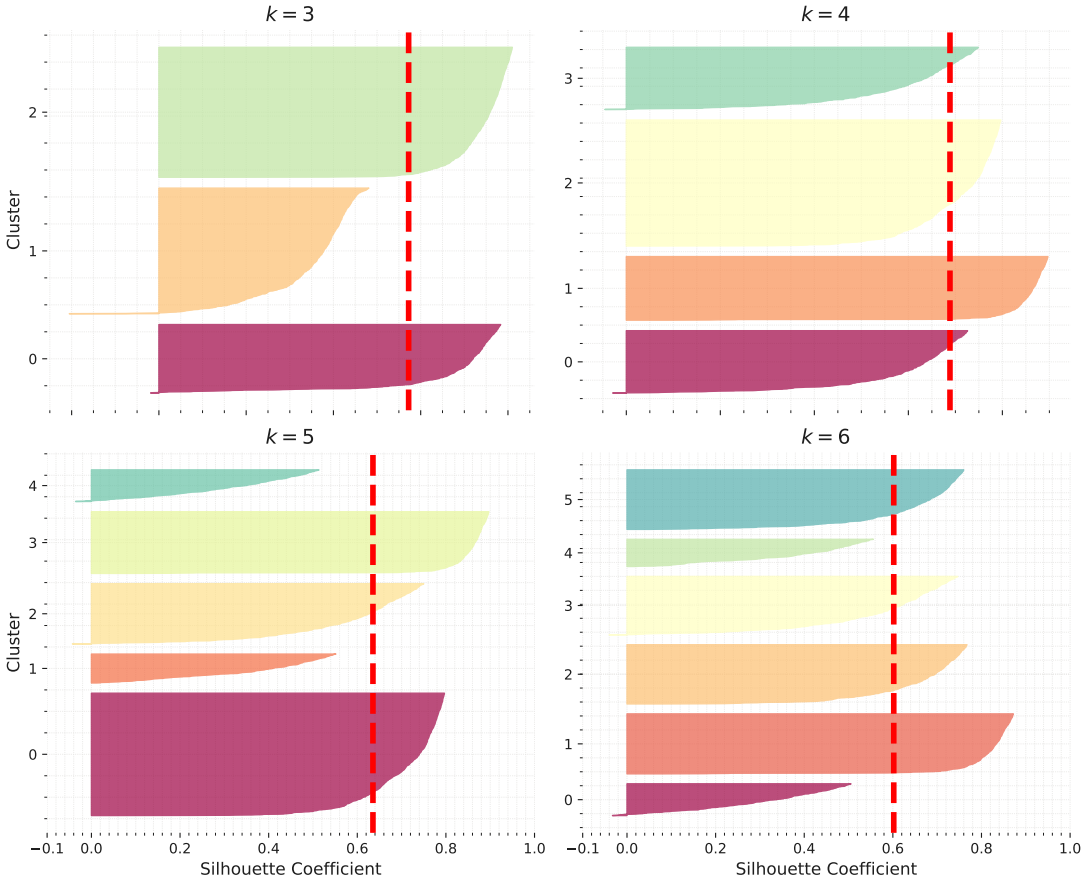

Figure 5.10: Analyzing the silhouette diagrams for various values of k.

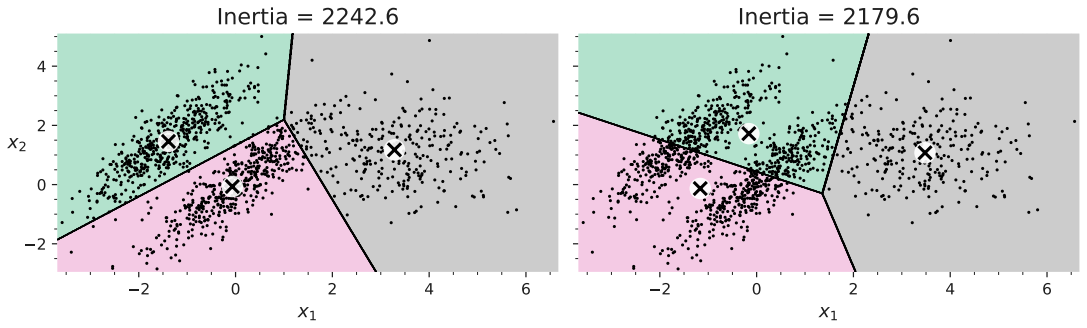

Figure 5.11: *k*-means fails to cluster these ellipsoidal blobs properly.

5.1.2 Limits of K-Means

Despite its many merits, most notably being fast and scalable, *k*-means is not perfect. As we saw, it is necessary to run the algorithm several times to avoid sub-optimal solutions, plus we need to specify the number of clusters, which can be quite a hassle. Moreover, *k*-means does not behave very well when the clusters have varying sizes, different densities, or non-spherical shapes. For example, **Fig.** 5.11 shows how *k*-means clusters a dataset containing three (3) ellipsoidal clusters of different dimensions, densities, and orientations.

As can be seen, neither of these solutions is any good. The solution on the left is better, but it still chops off 25% of the middle cluster and assigns it to the cluster on the right. The solution on the right is just terrible, even though its inertia is lower. So, depending on the data, different clustering algorithms may perform better. On these types of elliptical clusters, **Gaussian mixture models** work great.

> It is important to scale the input features before we run k-means, or the clusters may be very stretched and k-means will perform poorly. Scaling the features does not guarantee that all the clusters will be nice and spherical, but it generally helps k-means.

Now let's look at a few ways we can benefit from clustering and for these will use k-means

5.1.3 Using Clustering for Image Segmentation

Image segmentation is the task of partitioning an image into multiple segments. There are several variants: In color segmentation,

Colour Segmentation pixels with a similar color get assigned to the same segment. This is sufficient in many applications.

> For example, if we want to analyze satellite images to measure how much total forest area there is in a region, color segmentation may be just fine.

Semantic Segmentation all pixels that are part of the same object type get assigned to the same segment.

> For example, in a self-driving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the pedestrian segment.

Instance Segmentation all pixels that are part of the same individual object are assigned to the same segment. In this case there would be a different segment for each pedestrian.

The state of the art in semantic or instance segmentation today is achieved using complex architectures based on Convolutional Neural Networks (CNN) [14]. Here we are going to focus on the colour segmentation task, using k-means. We'll start by importing the Pillow package, which we'll then use to load the `Fruit.png` image (see the upper-left image in **Fig.** 5.12), assuming it's located at filepath:

[14] Which is taught in **B.Sc. Image processing**.

C.R. 13

```python
import PIL

image = np.asarray(PIL.Image.open(filepath))
print(image.shape)
```

```text
(680, 680, 3)
```

The image is represented as a 3D array. The first dimension's size is the height, the second is the width, and the third is the number of color channels, in this case red, green, and blue (RGB). In other words, for each pixel there is a 3D vector containing the intensities of red, green, and blue as unsigned 8- bit integers between 0 and 255. Some images may have fewer channels[15], and some images may have more

[15] such as grayscale images, which only have one.

channels (such as images with an additional alpha channel for transparency, or satellite images, which often contain channels for additional light frequencies (like infrared). The following code reshapes the array to get a long list of RGB colors, then it clusters these colours using k-means with eight clusters. It creates a `segmented_img` array containing the nearest cluster centre for each pixel (i.e., the mean colour of each pixel's cluster), and lastly it reshapes this array to the original image shape. The third line uses advanced NumPy indexing; for example, if the first 10 labels in `kmeans_.labels_` are equal to 1, then the first 10 colors in `segmented_img` are equal to `kmeans.cluster_centers_[1]`:

C.R. 14
python
```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8, n_init=10, random_state=42).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)

segmented_imgs = []
n_colors = (10, 8, 6, 4, 2)

for n_clusters in n_colors:
    kmeans = KMeans(n_clusters=n_clusters, n_init=10, random_state=42).fit(X)
    segmented_img = kmeans.cluster_centers_[kmeans.labels_]
    segmented_imgs.append(segmented_img.reshape(image.shape))

```

This outputs the image shown in the upper right of **Fig.** 5.12 . We can experiment with various numbers of clusters, as shown in the figure. When we use fewer than eight clusters, notice that the ladybug's flashy red color fails to get a cluster of its own: it gets merged with colors from the environment. This is because k-means prefers clusters of similar sizes. The ladybug is small—much smaller than the rest of the image—so even though its colour is flashy, k-means fails to dedicate a cluster to it.

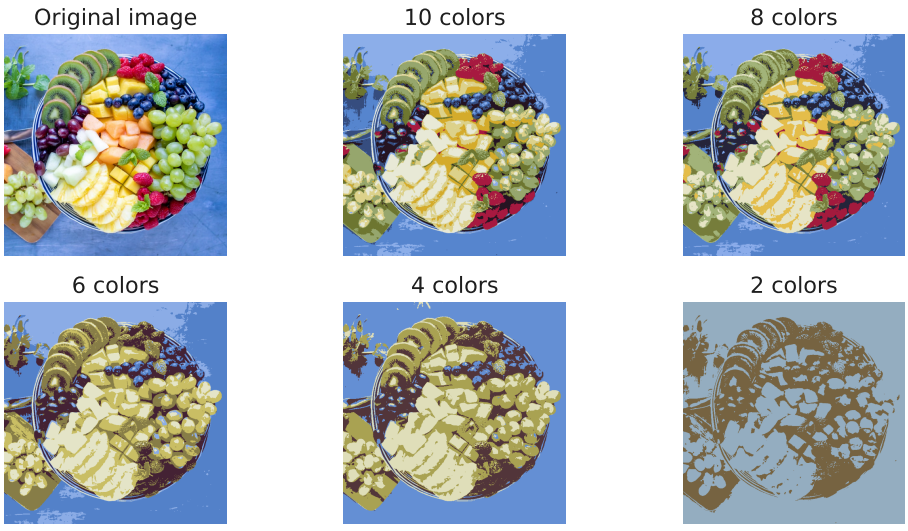

Figure 5.12: Image segmentation using k-means with various numbers of color clusters

Now this looks very pretty. Now it is time to look at another application of clustering.

5.1.4 Using Clustering for Semi-Supervised Learning

Another use case for clustering is in semi-supervised learning, when we have plenty of unlabelled instances and very few labelled instances. In this section, we'll use the digits dataset, which is a simple Modified National Institute of Standards and Technology (MNIST)-like dataset containing 1,797 grayscale 8-by-8 images representing the digits 0 to 9. First, let's load and split the dataset (it's already shuffled):

C.R. 15 python
```python
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
X_train, y_train = X_digits[:1400], y_digits[:1400]
X_test, y_test = X_digits[1400:], y_digits[1400:]
```

We will pretend we only have labels for 50 instances. To get a baseline performance, let's train a logistic regression model on these 50 labelled instances:

C.R. 16 python
```python
from sklearn.linear_model import LogisticRegression

n_labeled = 50
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

We can then measure the accuracy of this model on the test set:

> The test set must be labelled:

C.R. 17 python
```python
log_reg.score(X_test, y_test)
```

text
```
0.7581863979848866
```

The model's accuracy is just 75.8%. That's not great: indeed, if we try training the model on the full training set, we will find that it will reach about 90.9% accuracy. Let's see how we can do better. First, let's cluster the training set into 50 clusters. Then, for each cluster, we'll find the image closest to the centroid. We'll call these images the representative images where we can see 50 of them in **Fig.** 5.13

C.R. 18 python
```python
k = 50
kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = X_digits_dist.argmin(axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Let's look at each image and manually label them:

C.R. 19 python
```python
y_representative_digits = np.array([
    8, 4, 9, 6, 7, 5, 3, 0, 1, 2,
    3, 3, 4, 7, 2, 1, 5, 1, 6, 4,
    5, 6, 5, 7, 3, 1, 0, 8, 4, 7,
```

Figure 5.13: Fifty representative digit images (one per cluster).

C.R. 20
```
    1, 1, 8, 2, 9, 9, 5, 9, 7, 4,
    4, 9, 7, 8, 2, 6, 6, 3, 2, 8
])
```
python

Now we have a dataset with just 50 labelled instances, but instead of being random instances, each of them is a representative image of its cluster. Let's see if the performance is any better:

C.R. 21
```
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_representative_digits, y_representative_digits)
log_reg.score(X_test, y_test)
```
python

```
0.8387909319899244
```
text

Wow! We jumped from 75.8% accuracy to 83.8%, although we are still only training the model on 50 instances. Since it is often costly and painful to label instances, especially when it has to be done manually by experts, it is a good idea to label representative instances rather than just random instances. But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster? This is called label propagation:

C.R. 22
```
y_train_propagated = np.empty(len(X_train), dtype=np.int64)
for i in range(k):
    y_train_propagated[kmeans.labels_ == i] = y_representative_digits[i]
```
python

Now let's train the model again and look at its performance:

C.R. 23
```
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train, y_train_propagated)
```
python

C.R. 24
```
log_reg.score(X_test, y_test)
```
python

```
0.8589420654911839
```
text

We got another significant accuracy boost.

Active Learning

Active Learning

To continue improving our model and our training set, the next step could be to do a few rounds of active learning, which is when a human expert interacts with the learning algorithm, providing labels for specific instances when the algorithm requests them. There are many different strategies for active learning, but one of the most common ones is called uncertainty sampling. Here is how it works:

1. The model is trained on the labelled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
2. The instances for which the model is most uncertain(i.e.,where its estimated probability is lowest) are given to the expert for labelling.
3. We iterate this process until the performance improvement stops being worth the labeling effort.

Other active learning strategies include labeling the instances that would result in the largest model change or the largest drop in the model's validation error, or the instances that different models disagree on (e.g., an SVM and a random forest).

Before we move on to Gaussian mixture models, let's take a look at Density-Based Spatial Clustering of Applications with Noise (DBSCAN), another popular clustering algorithm that illustrates a very different approach based on local density estimation. This approach allows the algorithm to identify clusters of arbitrary shapes.

5.1.5 DBSCAN

The DBSCAN algorithm defines clusters as continuous regions of high density. Here is how it works:

1. For each instance, the algorithm counts how many instances are located within a small distance ϵ from it. This region is called the instance's ϵ-neighbourhood.
2. If an instance has at least `min_samples` instances in its ϵ-neighborhood (including itself), then it is considered a core instance. In other words, core instances are those that are located in dense regions.
3. All instances in the neighborhood of a core instance belong to the same cluster. This neighbourhood may include other core instances, therefore, a long sequence of neighboring core instances forms a single cluster.
4. Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

This algorithm works well if all the clusters are well separated by low-density regions. The DBSCAN class in `sklearn` is as simple to use as we might expect. Let's test it on the moons dataset, which is a toy dataset:

C.R. 25 python

```python
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05, random_state=42)
dbscan = DBSCAN(eps=0.05, min_samples=5)
```

```python
dbscan.fit(X)
```
C.R. 26

The labels of all the instances are now available in the `labels_` instance variable:

```python
print(dbscan.labels_[:10])
```
C.R. 27

```
[ 0  2 -1 -1  1  0  0  0  2  5]
```

Notice that some instances have a cluster index equal to -1, which means that they are considered as anomalies by the algorithm. The core instances indices are available in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

```python
print(dbscan.core_sample_indices_[:10])
```
C.R. 28

```
[ 0  4  5  6  7  8 10 11 12 13]
```

```python
print(dbscan.components_)
```
C.R. 29

```
[[-0.02137124  0.40618608]
 [-0.84192557  0.53058695]
 [ 0.58930337 -0.32137599]
 ...
 [ 1.66258462 -0.3079193 ]
 [-0.94355873  0.3278936 ]
 [ 0.79419406  0.60777171]]
```

This clustering is represented in the Left-hand Side (LHS) plot of **Fig.** 5.14 . As we can see, it identified quite a lot of anomalies, plus seven different clusters. Fortunately, if we widen each instance's neighbourhood by increasing eps to 0.2, we get the clustering on the right, which looks much better. Let's continue with this model.

Surprisingly, the DBSCAN class does not have a `predict()` method, although it has a `fit_predict()` method. In other words, it cannot predict which cluster a new instance belongs to. This decision was made because different classification algorithms can be better for different tasks, so the authors decided to let the user choose which one to use. Moreover, it's not hard to implement. For example, let's train a `KNeighborsClassifier`:

```python
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```
C.R. 30

Now, given a few new instances, we can predict which clusters they most likely belong to and even estimate a probability for each cluster:

C.R. 31
```python
X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
print(knn.predict(X_new))
```

```text
[1 0 1 0]
```

C.R. 32
```python
print(knn.predict_proba(X_new))
```

```text
[[0.18 0.82]
 [1.   0.  ]
 [0.12 0.88]
 [1.   0.  ]]
```

Note that we only trained the classifier on the core instances, but we could also have chosen to train it on all the instances, or all but the anomalies.

> This choice depends on the final task

The decision boundary is represented in **Fig.** 5.15 (the crosses represent the four instances in `X_new`). Notice that since there is no anomaly in the training set, the classifier always chooses a cluster, even when that cluster is far away. It is fairly straightforward to introduce a maximum distance, in which case the two instances that are far away from both clusters are classified as anomalies. To do this, use the `kneighbors()` method of the `KNeighborsClassifier`. Given a set of instances, it returns the distances and the indices of the k-nearest neighbours in the training set (two matrices, each with k columns):

C.R. 33
```python
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
print(y_pred.ravel())
```

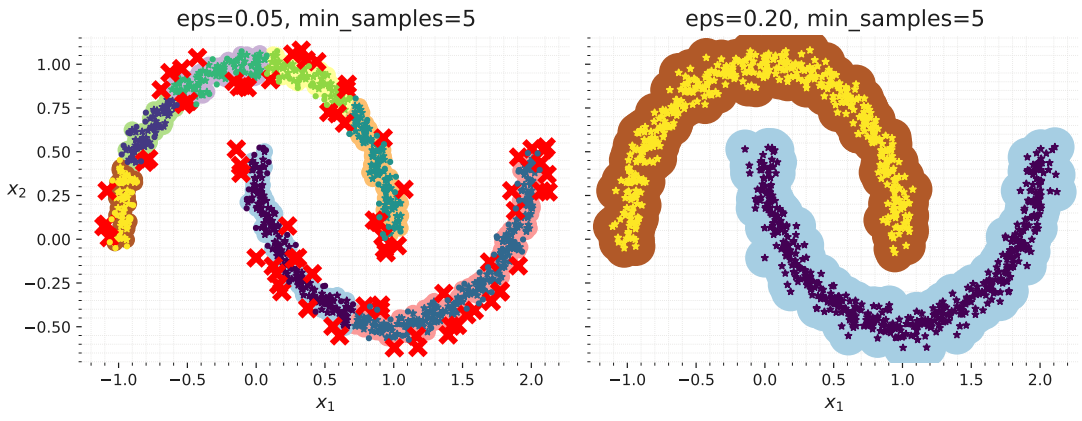

Figure 5.14: DBSCAN clustering using two different neighborhood radiuses.

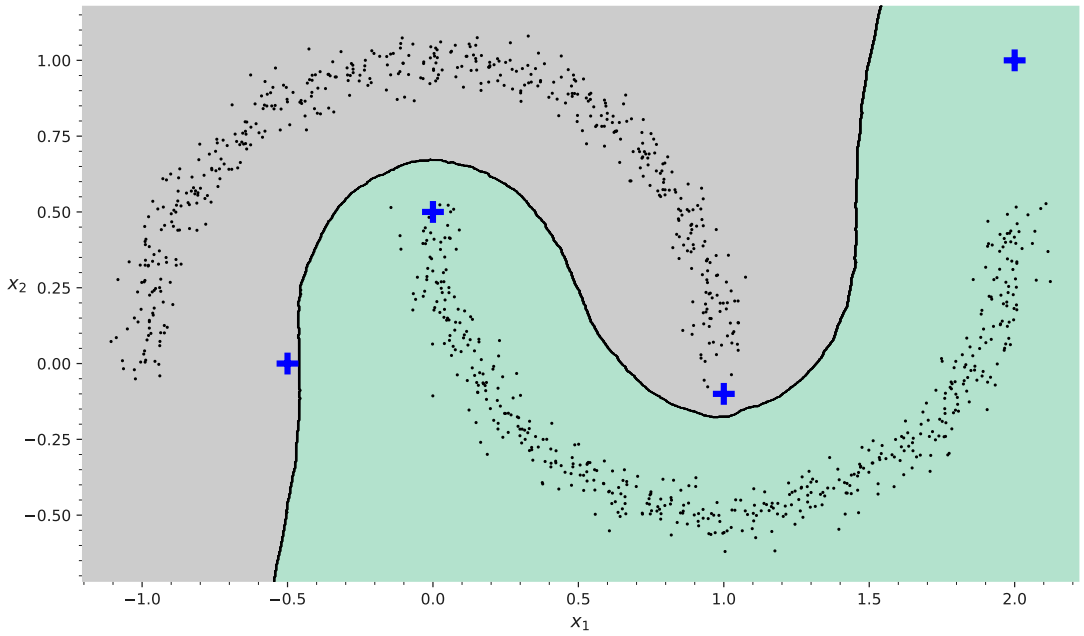

Figure 5.15: Decision boundary between two clusters

```
1 [-1  0  1 -1]                                                    text
```

In short, DBSCAN is a very simple yet powerful algorithm capable of identifying any number of clusters of any shape. It is robust to outliers, and it has just two hyperparameters (`eps` and `min_samples`). If the density varies significantly across the clusters, however, or if there's no sufficiently low- density region around some clusters, DBSCAN can struggle to capture all the clusters properly. Moreover, its computational complexity is roughly O(m2n), so it does not scale well to large datasets.

Other Clustering Algorithms

`sklearn` implements several more clustering algorithms that we should take a look at. Here is just some of them:

Agglomerative clustering A hierarchy of clusters is built from the bottom up. Think of many tiny bubbles floating on water and gradually attaching to each other until there's one big group of bubbles. Similarly, at each iteration, agglomerative clustering connects the nearest pair of clusters (starting with individual instances). If we drew a tree with a branch for every pair of clusters that merged, we would get a binary tree of clusters, where the leaves are the individual instances. This approach can capture clusters of various shapes; it also produces a flexible and informative cluster tree instead of forcing we to choose a particular cluster scale, and it can be used with any pairwise distance. It can scale nicely to large numbers of instances if we provide a connectivity matrix, which is a sparse m × m matrix that indicates which pairs of instances are neighbors (e.g., returned by `sklearn.neighbors.kneighbors_graph()`). Without a connectivity matrix, the algorithm does not scale well to large datasets.

BIRCH The balanced iterative reducing and clustering using hierarchies (BIRCH) algorithm was designed specifically for very large datasets, and it can be faster than batch k-means, with similar

results, as long as the number of features is not too large (<20). During training, it builds a tree structure containing just enough information to quickly assign each new instance to a cluster, without having to store all the instances in the tree: this approach allows it to use limited memory while handling huge datasets.

Mean-shift This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean. Next, it iterates this mean-shifting step until all the circles stop moving (i.e., until each of them is centered on the mean of the instances it contains). Mean-shift shifts the circles in the direction of higher density, until each of them has found a local density maximum. Finally, all the instances whose circles have settled in the same place (or close enough) are assigned to the same cluster. Mean-shift has some of the same features as DBSCAN, like how it can find any number of clusters of any shape, it has very few hyperparameters (just one—the radius of the circles, called the bandwidth), and it relies on local density estimation. But unlike DBSCAN, mean-shift tends to chop clusters into pieces when they have internal density variations. Unfortunately, its computational complexity is O(m2n), so it is not suited for large datasets.

Affinity Propagation In this algorithm, instances repeatedly exchange messages between one another until every instance has elected another instance (or itself) to represent it. These elected instances are called exemplars. Each exemplar and all the instances that elected it form one cluster. In real-life politics, we typically want to vote for a candidate whose opinions are similar to ours, but we also want them to win the election, so we might choose a candidate we don't fully agree with, but who is more popular. We typically evaluate popularity through polls. Affinity propagation works in a similar way, and it tends to choose exemplars located near the center of clusters, similar to k-means. But unlike with k-means, we don't have to pick a number of clusters ahead of time: it is determined during training. Moreover, affinity propagation can deal nicely with clusters of different sizes. Sadly, this algorithm has a computational complexity of O(m2), so it is not suited for large datasets.

Spectral clustering This algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (i.e., it reduces the matrix's dimensionality), then it uses another clustering algorithm in this low- dimensional space (`sklearn`'s implementation uses k-means). Spectral clustering can capture complex cluster structures, and it can also be used to cut graphs (e.g., to identify clusters of friends on a social network). It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

Now let's dive into Gaussian mixture models, which can be used for density estimation, clustering, and anomaly detection.

5.2 Gaussian Mixtures

A Gaussian Mixture Model (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown. All the instances generated from a single Gaussian distribution form a cluster that typically looks like an ellipsoid. Each cluster can have a different ellipsoidal shape, size, density, and orientation, just like in **Fig.** 5.7 . When we observe an instance, we know it was generated from one of the Gaussian distributions,

but we are not told which one, and we do not know what the parameters of these distributions are.

There are several GMM variants. In the simplest variant, implemented in the `GaussianMixture` class, we must know in advance the number k of Gaussian distributions. The dataset X is assumed to have been generated through the following probabilistic process:

- For each instance, a cluster is picked randomly from among k clusters. The probability of choosing the j_{th} cluster is the cluster's weight $\phi^{(j)}$. The index of the cluster chosen for the j_{th} instance is noted $z^{(i)}$.
- If the i^{th} instance was assigned to the j_{th} cluster (i.e., $z^{(i)} = j$), then the location $\mathbf{x}^{(i)}$ of this instance is sampled randomly from the Gaussian distribution with mean $\mu^{(j)}$ and covariance matrix $\Sigma^{(j)}$. This is noted as:

$$\mathbf{x}^{(i)} \sim \mathcal{N}\left(\mu^{(j)}, \Sigma^{(j)}\right)$$

So what can we do with such a model? Well, given the dataset **X**, we typically want to start by estimating the weights ϕ and all the distribution parameters $\mu^{(1)}$ to $\mu^{(k)}$ and $\Sigma^{(1)}$ to $\Sigma^{(k)}$. `sklearn`'s `GaussianMixture` class makes this super easy:

C.R. 34 python
```python
from sklearn.mixture import GaussianMixture
```

C.R. 35 python
```python
gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

Let's look at the parameters that the algorithm estimated:

C.R. 36 python
```python
print(gm.weights_)
```

text
```
[0.40005972 0.20961444 0.39032584]
```

C.R. 37 python
```python
print(gm.means_)
```

text
```
[[-1.40764129  1.42712848]
 [ 3.39947665  1.05931088]
 [ 0.05145113  0.07534576]]
```

Great, it worked fine! Indeed, two of the three clusters were generated with 500 instances each, while the third cluster only contains 250 instances. So the true cluster weights are 0.4, 0.2, and 0.4, respectively, and that's roughly what the algorithm found. Similarly, the true means and covariance matrices are quite close to those found by the algorithm. But how? This class relies on the Expectation Minimisation (EM) algorithm, which has many similarities with the k-means algorithm where it also initializes the cluster parameters randomly, then it repeats two steps until convergence, first assigning instances to clusters (this is called the expectation step) and then updating the clusters (this is called the maximisation step). In the context of clustering, we can think of EM as a generalisation of k-means that not only finds the cluster centres ($\mu^{(1)}$ to $\mu^{(k)}$), but also their size, shape, and orientation ($\Sigma^{(1)}$ to $\Sigma^{(k)}$), as well as their relative weights ($\phi^{(1)}$ to $\phi^{(j)}$). Unlike k-means, though, EM uses soft cluster

assignments, not hard assignments. For each instance, during the expectation step, the algorithm estimates the probability that it belongs to each cluster (based on the current cluster parameters). Then, during the maximisation step, each cluster is updated using all the instances in the dataset, with each instance weighted by the estimated probability that it belongs to that cluster. These probabilities are called the responsibilities of the clusters for the instances. During the maximization step, each cluster's update will mostly be impacted by the instances it is most responsible for.

> Unfortunately, just like k-means, EM can end up converging to poor solutions, so it needs to be run several times, keeping only the best solution. This is why we set `n_init` to 10. By default `n_init` is set to 1.

We can check whether or not the algorithm converged and how many iterations it took:

C.R. 38
```python
print(gm.converged_)
```

```text
True
```

C.R. 39
```python
print(gm.n_iter_)
```

```text
4
```

Now that we have an estimate of the location, size, shape, orientation, and relative weight of each cluster, the model can easily assign each instance to the most likely cluster (hard clustering) or estimate the probability that it belongs to a particular cluster (soft clustering). Just use the `predict()` method for hard clustering, or the `predict_proba()` method for soft clustering:

C.R. 40
```python
print(gm.predict(X))
```

```text
[2 2 0 ... 1 1 1]
```

C.R. 41
```python
print(gm.predict_proba(X).round(3))
```

```text
[[0.    0.023 0.977]
 [0.001 0.016 0.983]
 [1.    0.    0.   ]
 ...
 [0.    1.    0.   ]
 [0.    1.    0.   ]
 [0.    1.    0.   ]]
```

A Gaussian mixture model is a generative model, meaning we can sample new instances from it (note that they are ordered by cluster index):

C.R. 42
```python
X_new, y_new = gm.sample(6)
print(X_new)
```

```text
[[-2.32491052  1.04752548]
 [-1.16654983  1.62795173]
 [ 1.84860618  2.07374016]
 [ 3.98304484  1.49869936]
 [ 3.8163406   0.53038367]
 [ 0.38079484 -0.56239369]]
```

C.R. 43
```python
print(y_new)
```

```text
[0 0 1 1 1 2]
```

It is also possible to estimate the density of the model at any given location. This is achieved using the `score_samples()` method: for each instance it is given, this method estimates the log of the Probability Density Function (PDF) at that location. The greater the score, the higher the density:

C.R. 44
```python
print(gm.score_samples(X).round(2))
```

```text
[-2.61 -3.57 -3.33 ... -3.51 -4.4  -3.81]
```

If we compute the exponential of these scores, we get the value of the PDF at the location of the given instances. These are not probabilities, but probability densities: they can take on any positive value, not just a value between 0 and 1. To estimate the probability that an instance will fall within a particular region, we would have to integrate the PDF over that region (if we do so over the entire space of possible instance locations, the result will be 1). **Fig.** 5.16 shows the cluster means, the

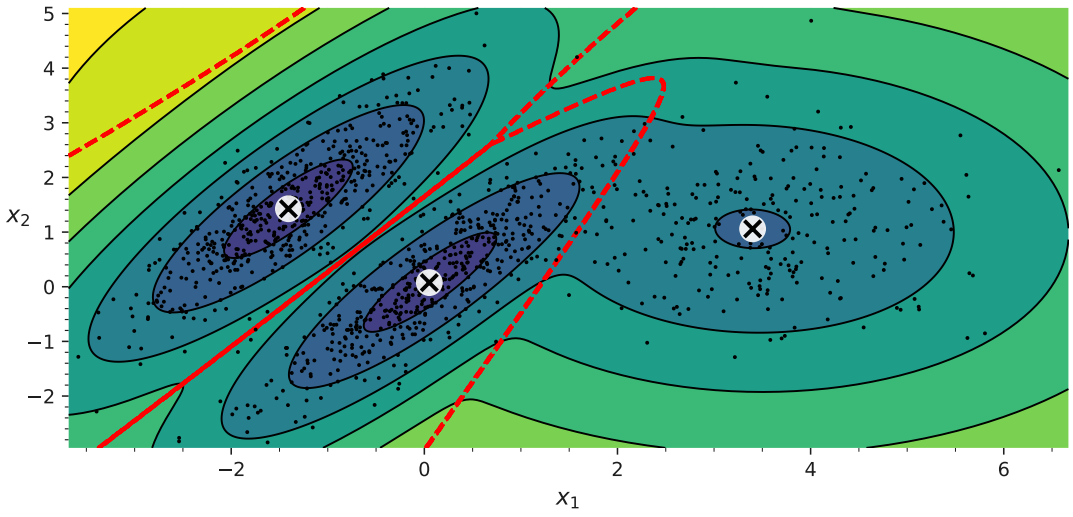

Figure 5.16: Cluster means, decision boundaries, and density contours of a trained Gaussian mixture model.

decision boundaries (dashed lines), and the density contours of this model.

The algorithm clearly found an excellent solution. Of course, we made its task easy by generating the data using a set of 2D Gaussian distributions[16]. 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. We 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, set the `covariance_type` hyperparameter to one of the following values:

[16] unfortunately, real-life data is not always so Gaussian and low-dimensional

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). **Fig.** 5.17 plots the solutions found by the EM algorithm when `covariance_type` is set to `"tied"` or `"spherical"`.

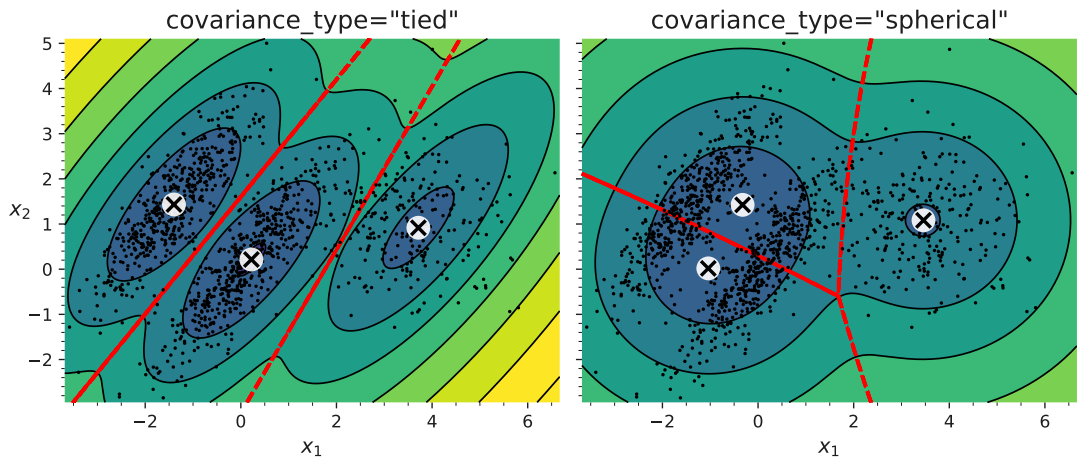

Figure 5.17: Gaussian mixtures for tied clusters (left) and spherical clusters (right)

Computational Complexity

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(kmn2 + kn3), so it will not scale to large numbers of features.

Gaussian mixture models can also be used for anomaly detection. We'll see how in the next section.

5.2.1 Using Gaussian Mixtures for Anomaly Detection

Using a Gaussian mixture model for anomaly detection is quite simple.

> Any instance located in a low-density region can be considered an anomaly.

We must define what density threshold we 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 2%. We then set the density threshold to be the value that results in having 2% of the instances located in areas below that threshold density. If we notice that we get too many false positives (i.e., perfectly good products that are flagged as defective), we can lower the threshold. Conversely, if we have too many false negatives (i.e., defective products that the system does not flag as defective), we can increase the threshold. This is the usual precision/recall trade-off. Here is how we would identify the outliers using the fourth percentile lowest density as the threshold (i.e., approximately 4% of the instances will be flagged as anomalies):

C.R. 45
python
```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 2)
anomalies = X[densities < density_threshold]
```

Fig. 5.18 represents these anomalies as stars. 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 used to clean up a dataset.

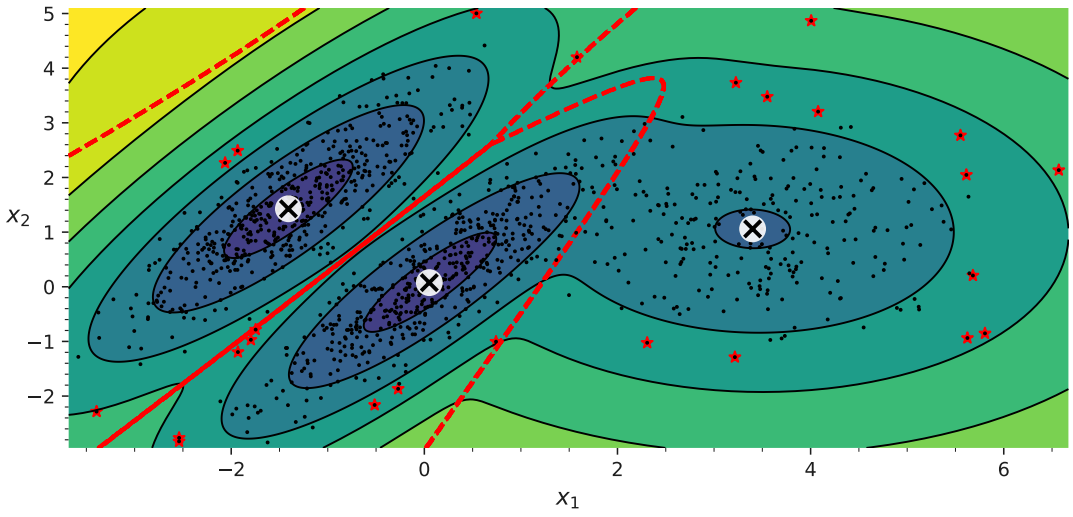

Figure 5.18: Anomaly detection using a Gaussian mixture model.

Working with Outliers

Gaussian mixture models try to fit all the data, including the outliers; if we have too many of them this will bias the model's view of "normality", and some outliers may wrongly be considered as normal. If this happens, we 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.

Just like k-means, the `GaussianMixture` algorithm requires we to specify the number of clusters. So how can we find that number?

5.2.2 Selecting the Number of Clusters

With k-means, we can 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 as they are not reliable when the clusters are not spherical or have different sizes. Instead, we can try to find the model that minimises a theoretical information criterion, such as the Bayesian Information Criterion (BIC) or the Akaike Information Criterion (AIC), defined as:

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

where m is the number of instances, p is the number of parameters learned by the model and $\hat{L}$ is the maximised value of the likelihood function of the model. Both the BIC and AIC penalise models with more parameters to learn (e.g., more clusters) and reward models that fit the data well. They often end up selecting the same model. When they differ, the model selected by the BIC tends to be simpler (fewer parameters) than the one selected by the AIC, but tends to 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 everyday language, but have very different meanings in statistics. Given a statistical model with some parameters θ, the word "probability" is used to describe how plausible a future outcome x is (knowing the parameter values θ), while the word "likelihood" is used to describe how plausible a particular set of parameter values θ are, after the outcome x is known. Consider a 1D 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 9-19 shows the entire model $g(x;\, \theta)$ as a function of both x and θ. To estimate the probability distribution of a future outcome x, we need to set the model parameter θ. For example, if we set θ to 1.3 (the horizontal line), we get the probability density function f(x; θ=1.3) shown in the lower-left plot. Say we want to estimate the probability that x will fall between -2 and +2. We must calculate the integral of the PDF on this range (i.e., the surface of the shaded region). But what if we don't know θ, and instead if we have observed a single instance x=2.5 (the vertical line in the upper-left plot)? In this case, we get the likelihood function $L(\theta|x = 2.5) = f(x = 2.5; \theta)$, 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 we integrate a probability distribution over all possible values of x, we always get 1, but if we integrate the likelihood function over all possible values of θ the result can be any positive value. Given a dataset **X**, a common task is to try to estimate the most likely values for the model parameters. To do this, we must find the values that maximize the likelihood function, given X. In this example, if we have observed a

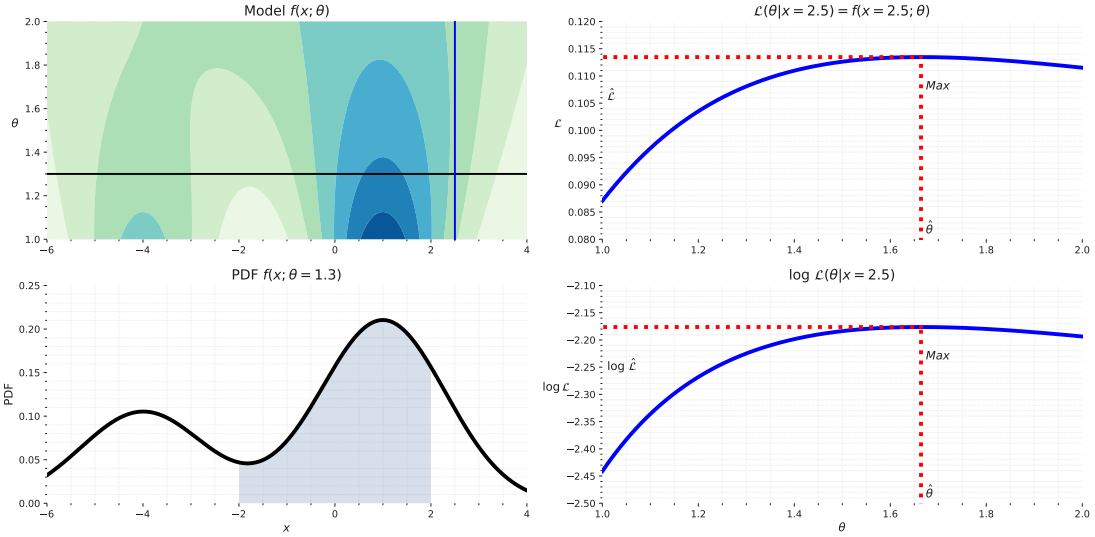

Figure 5.19: A model's parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right), and a log likelihood function (lower right).

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 maximising $\mathcal{L}(\theta|x)\, g(\theta)$ rather than just maximising $\mathcal{L}(\theta|x)$. This is called Maximum a-Posteriori (MAP) estimation. Since MAP constrains the parameter values, we can think of it as a regularised version of MLE. Notice that maximising the likelihood function is equivalent to maximising its logarithm (represented in the lower-right plot in **Fig.** 5.19). Indeed, the logarithm is a strictly increasing function, so if θ maximises the log-likelihood, it also maximises the likelihood. It turns out that it is generally easier to maximize the log likelihood. For example, if we observed several independent instances x(1) to x(m), we would need to find the value of θ that maximises 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 we have estimated $\hat{\theta}$, the value of θ that maximises the likelihood function, then we are ready to compute $\mathbf{L} = \mathcal{L}\left(\hat{\theta}, \mathbf{X}\right)$, which is the value used to compute the AIC and BIC; we can think of it as a measure of how well the model fits the data.

To compute the BIC and AIC, call the `bic()` and `aic()` 1methods:

Figure 9-20 shows the BIC for different numbers of clusters k. As we can see, both the BIC and the AIC are lowest when k=3, so it is most likely the best choice.

Bayesian Gaussian Mixture Models Rather than manually searching for the optimal number of clusters, we can use the BayesianGaussianMixture class, which is capable of giving weights equal (or close) to zero to unnecessary clusters. Set the number of clusters `n_components` to a value that we 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:

Perfect: the algorithm automatically detected that only three clusters are needed, and the resulting clusters are almost identical to the ones in Figure 9- 16. A final note about Gaussian mixture models: although they work great on clusters with ellipsoidal shapes, they don't do so well with clusters of very different shapes. For example, let's see what happens if we use a Bayesian Gaussian mixture model to cluster the moons dataset (see Figure 9-21). Oops! The algorithm desperately searched for ellipsoids,

so it found eight different clusters instead of two. 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. To conclude this chapter, let's take a quick look at a few algorithms capable of dealing with arbitrarily shaped clusters.

5.2.3 Other Algorithms for Anomaly and Novelty Detection

`sklearn` implements other algorithms dedicated to anomaly detection or novelty detection:

Fast-MCD Stands for minimum covariance determinant. Implemented by the EllipticEnvelope class, this algorithm is useful for outlier detection, in particular to clean up a dataset. It assumes that the normal instances (inliers) are generated from a single Gaussian distribution (not a mixture). It also assumes that the dataset is contaminated with outliers that were not generated from this Gaussian distribution. When the algorithm 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 technique gives a better estimation of the elliptic envelope and thus makes the algorithm 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 values) 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. Anomalies are usually far from other instances, so on average (across all the decision trees) they tend to get isolated in fewer 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 like all SVMs it does not scale to large datasets.

PCA and other dimensionality reduction techniques with an `inverse_transform()` method If we compare the reconstruction error of a normal instance with the reconstruction error of an anomaly, the latter will usually be much larger. This is a simple and often quite efficient anomaly detection approach.

Glossary

AIC Akaike Information Criterion. 3, 99

BIC Bayesian Information Criterion. 3, 99

CNN Convolutional Neural Networks. 3, 85

DBSCAN Density-Based Spatial Clustering of Applications with Noise. 3, 89, 90, 92

DT Decision Tree. 3, 5, 19–27, 29

EM Expectation Minimisation. 3, 94, 95, 97

GMM Gaussian Mixture Model. 3, 93

LHS Left-hand Side. 3, 90

LLE Locally Linear Embedding. 3, 52

LLN Law of Large Numbers. 3, 33

MAP Maximum a-Posteriori. 3, 100

ML Machine Learning. 3, 7, 11, 12, 14, 19, 22, 24, 32, 50, 51, 71, 72

MLE Maximum Likelihood Estimate. 3, 100

MNIST Modified National Institute of Standards and Technology. 3, 87

OoB Out-of-Bag. 3, 5, 31, 37, 38

PC Principal Component. 3, 57, 58

PCA Principal Component Analysis. 3, 52, 56–63, 65–67

PCM Pulse Code Modulation. 3, 74

PDF Probability Density Function. 3, 96, 99

RBF Radial Basis Function. 3, 76

RF Random Forest. 3, 32, 38–40, 48–50

SVD Singular Value Decomposition. 3, 57, 62

SVM Support Vector Machines. 3, 5, 7–13, 15–17, 33, 89

Bibliography

[1] David Arthur and Sergei Vassilvitskii. *k-means++: The advantages of careful seeding*. Tech. rep. Stanford, 2006.

[2] Eric Bair. "Semi-supervised clustering methods". In: *Wiley Interdisciplinary Reviews: Computational Statistics* 5.5 (2013), pp. 349–361.

[3] Richard Bellman. "Dynamic programming princeton university press". In: *Princeton, NJ* (1957), pp. 4–9.

[4] SM Aqil Burney and Humera Tariq. "K-means cluster analysis for image segmentation". In: *International Journal of Computer Applications* 96.4 (2014).

[5] Charles Elkan. "Using the triangle inequality to accelerate k-means". In: *Proceedings of the 20th international conference on Machine Learning (ICML-03)*. 2003, pp. 147–153.

[6] Edward W Forgy. "Cluster analysis of multivariate data: efficiency versus interpretability of classifications". In: *biometrics* 21 (1965), pp. 768–769.

[7] Alan Julian Izenman. "Introduction to manifold learning". In: *Wiley Interdisciplinary Reviews: Computational Statistics* 4.5 (2012), pp. 439–446.

[8] Tushar Kansal et al. "Customer segmentation using K-means clustering". In: *2018 international conference on computational techniques, electronics and mechanical systems (CTEMS)*. IEEE. 2018, pp. 135–139.

[9] Rashmi Kumari et al. "Anomaly detection in network traffic using K-mean clustering". In: *2016 3rd international conference on recent advances in information technology (RAIT)*. IEEE. 2016, pp. 387–393.

[10] Stuart Lloyd. "Least squares quantization in PCM". In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137.

[11] Rashmiranjan Nayak, Umesh Chandra Pati, and Santos Kumar Das. "A comprehensive review on deep learning-based methods for video anomaly detection". In: *Image and Vision Computing* 106 (2021), p. 104078.

[12] David Sculley. "Web-scale k-means clustering". In: *Proceedings of the 19th international conference on World wide web*. 2010, pp. 1177–1178.

[13] Oren Eli Zamir. *Clustering web documents: a phrase-based method for grouping search engine results*. University of Washington, 1999.

[14] Wang Zhiqiang and Liu Jun. "A review of object detection based on convolutional neural network". In: *2017 36th Chinese control conference (CCC)*. IEEE. 2017, pp. 11104–11109.

Github Link