Lecture Book
M.Sc Data Science II

D. T. McGuiness, PhD

version 2024.1

(2024, D. T. McGuiness, PhD)

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

1.1 Introduction

A SVM is a powerful Machine Learning (ML) model, capable of performing **linear or non-linear 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.

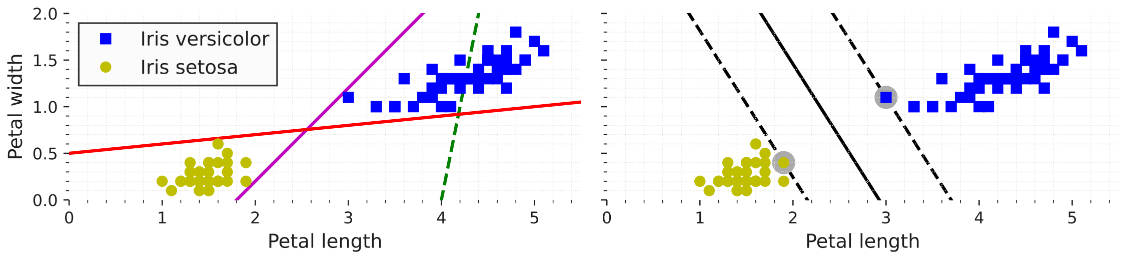

Figure 1.1: Large Margin Classifier

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

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:

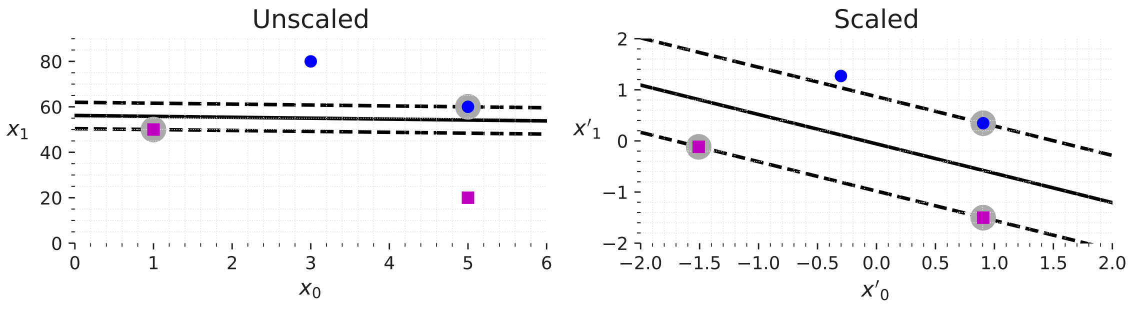

Figure 1.2: Sensitivity to feature scales.

- It only works if the data is linearly separable.
- 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.

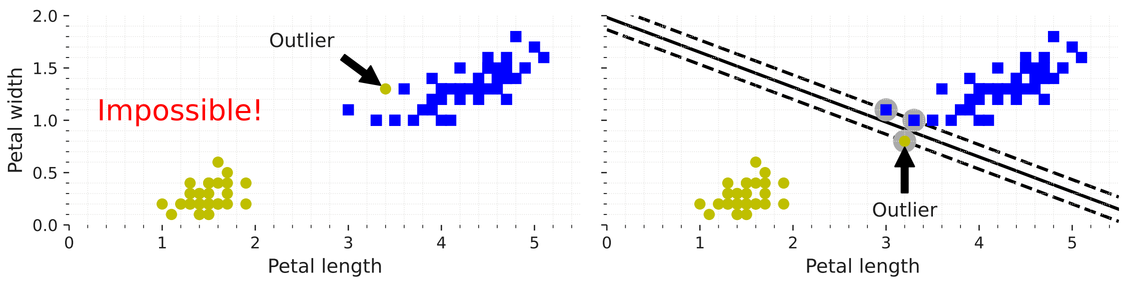

Figure 1.3: Hard margin sensitivity to outliers.

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`.

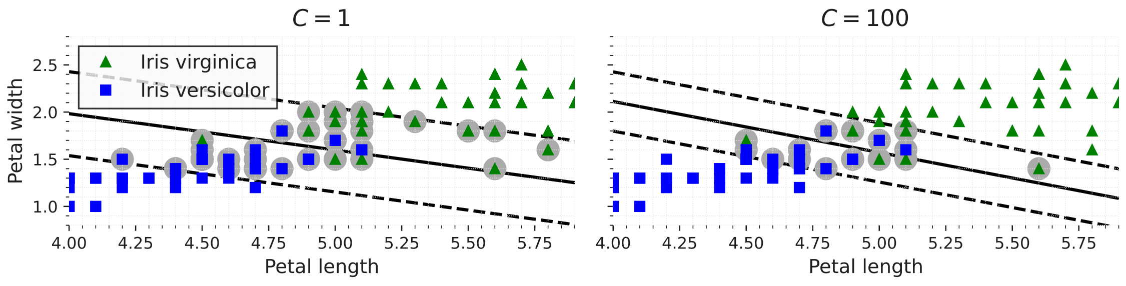

Figure 1.4: Large margin (left) v. fewer margin violations (right).

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`:

```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:

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

```
[ 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:

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

```
[ 0.66163816 -0.22035761]
```

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

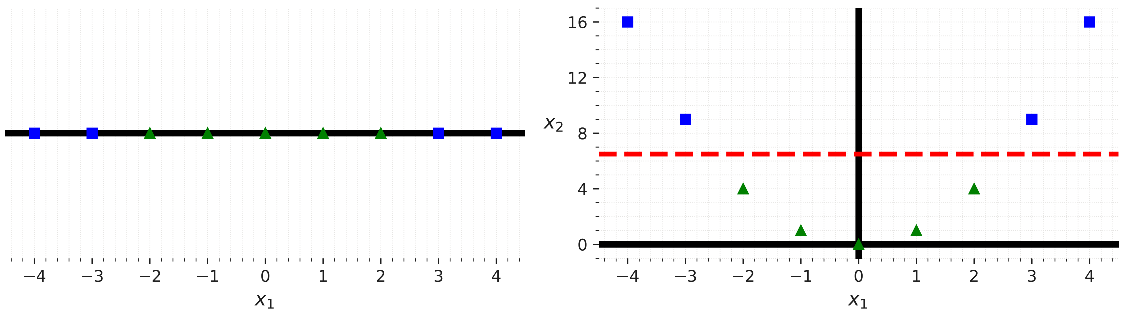

Figure 1.5: Adding features to make a dataset linearly separable.

pipeline containing a `PolynomialFeatures` 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:

```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)
```

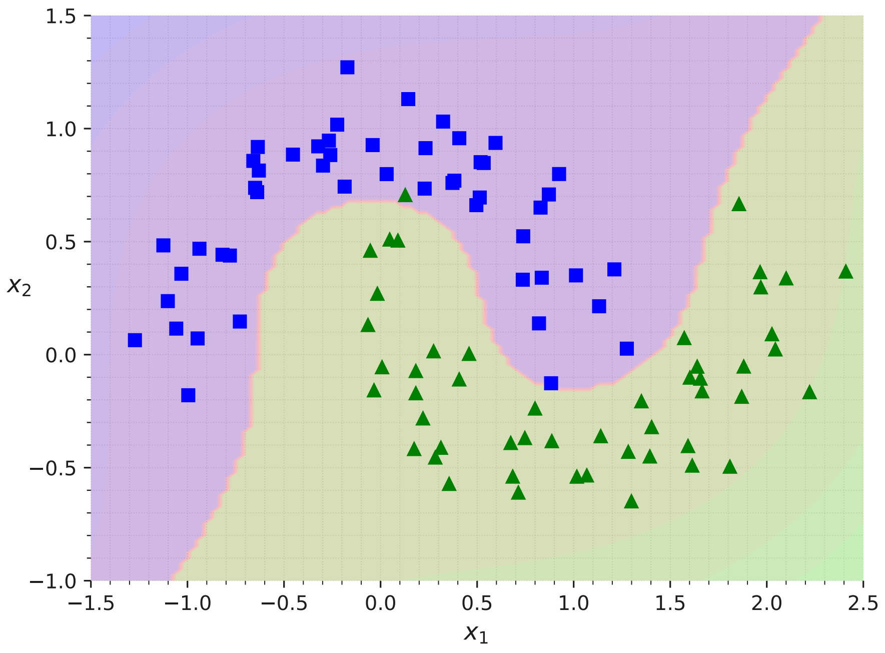

Figure 1.6: Linear SVM Classifier using polynomial features.

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:

```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 10th 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.

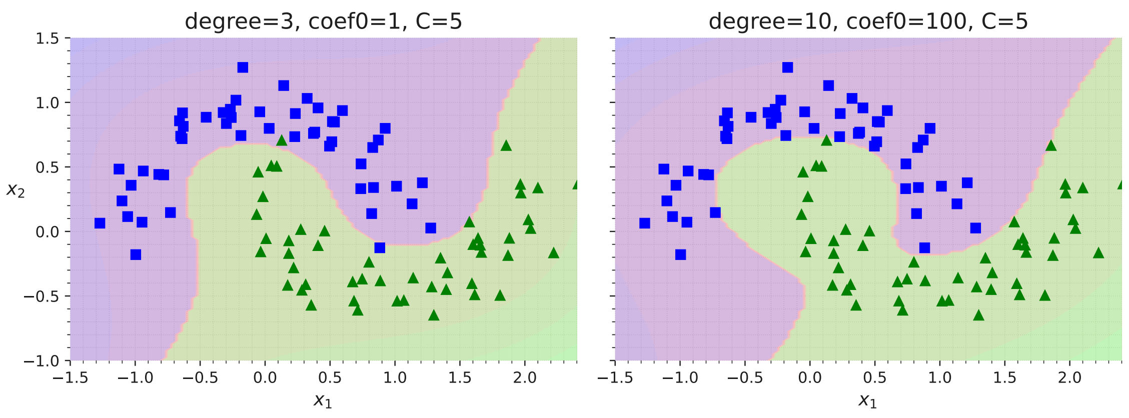

Figure 1.7: SVM classifiers with a polynomial kernel.

> 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

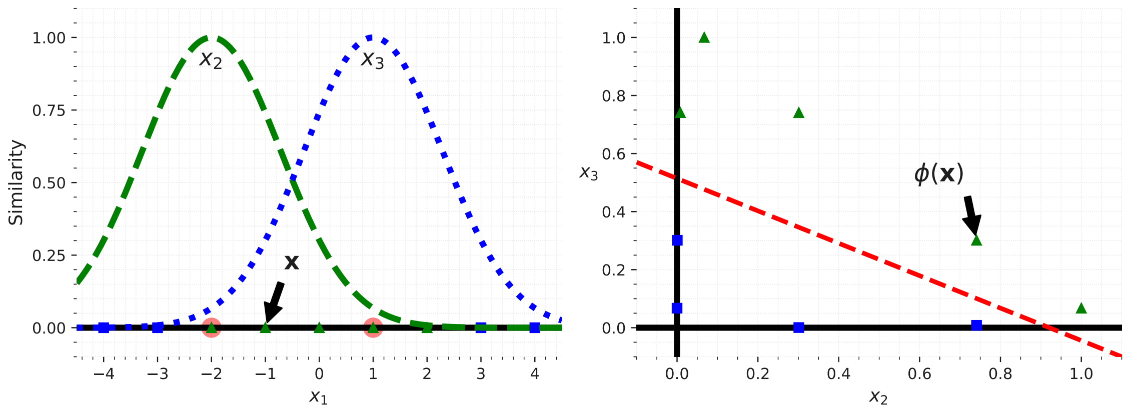

Figure 1.8: Similarity features using the Gaussian RBF.

$x_1 = 2$ and $x_1 = 1$ (see the left plot 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:

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

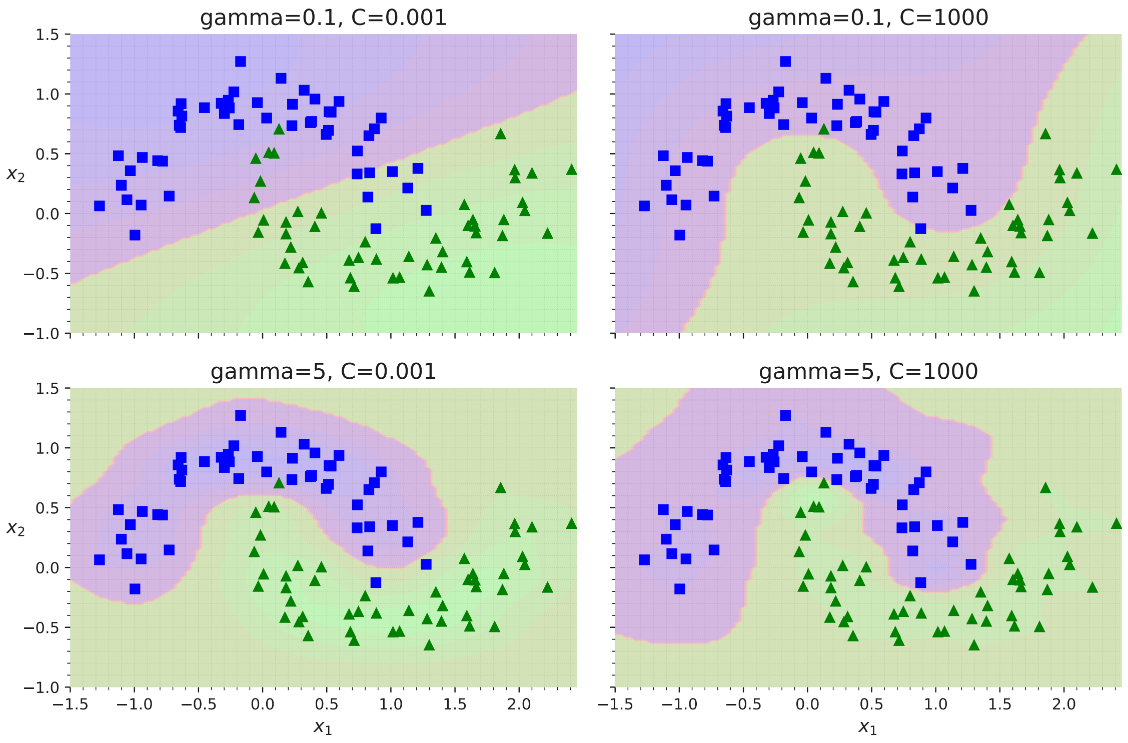

Figure 1.9: SVM classifiers using an RBF kernel.

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 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.

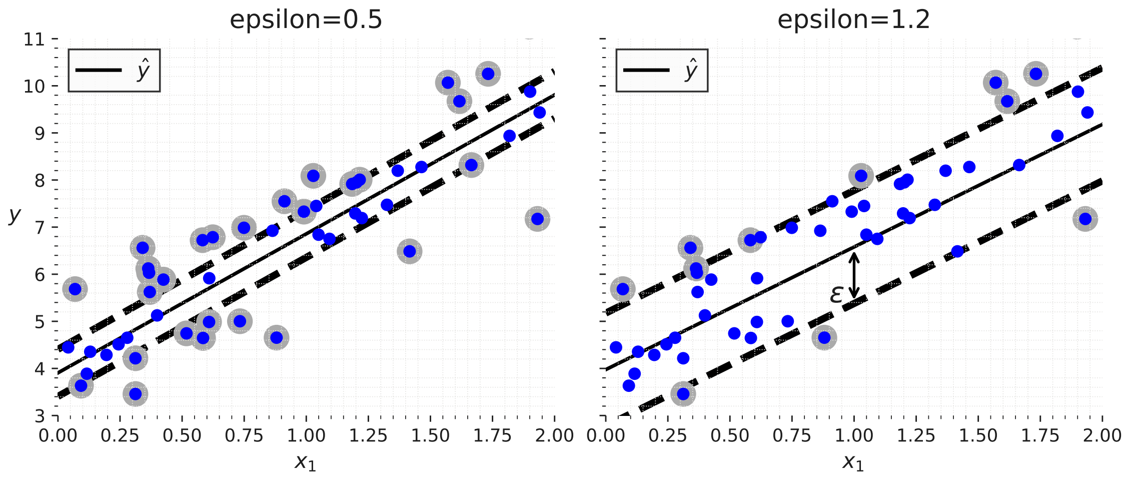

Figure 1.10: SVM regression.

```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 model represented on the left in Figure 1.11:

```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

```

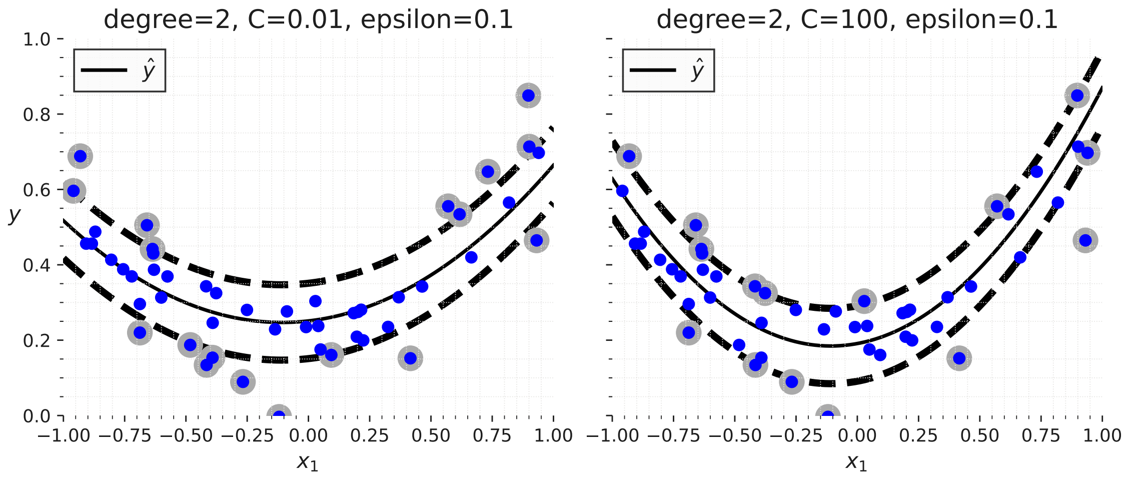

Figure 1.11: SVM regression using a second-degree polynomial kernel.

```python
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

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:

```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`:

```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:

```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.

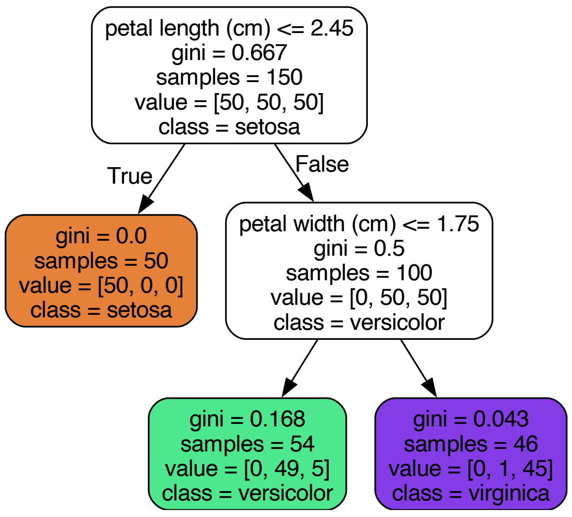

Figure 2.1

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 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 $1(0/54)2(49/54)2(5/54)2 \approx 0.168$.

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

where G_i is the Gini impurity of the i^{th} node, $p_{\text{i,k}}$ 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).

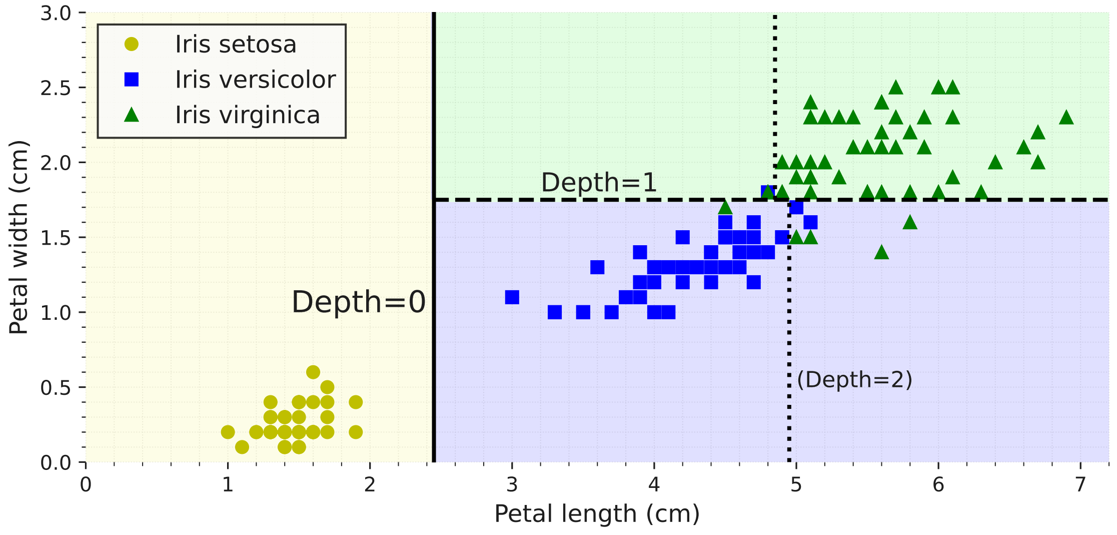

Figure 2.2: DT decision boundaries

> 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 explain 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:

```
print(tree_clf.predict_proba([[5, 1.5]]).round(3))
print(tree_clf.predict([[5, 1.5]]))
```

```
[[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 ≤ 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:

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

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:

```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:

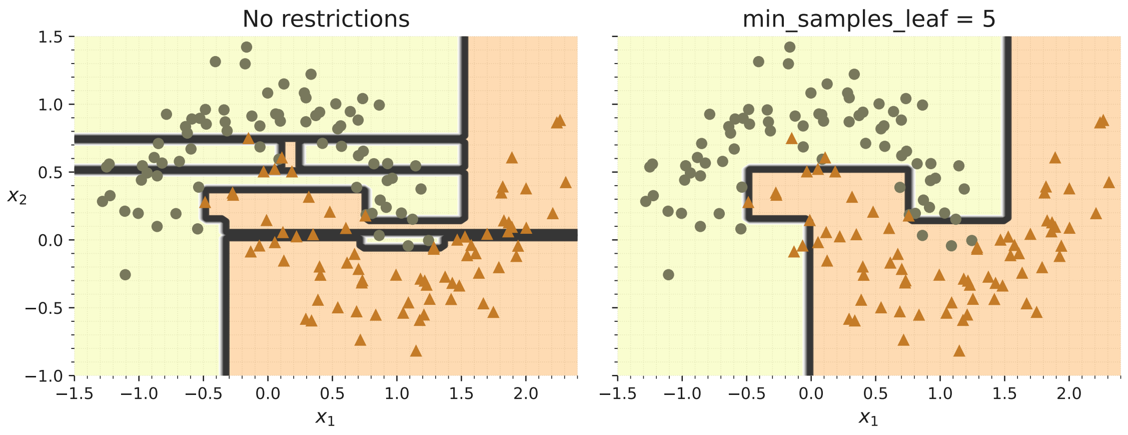

Figure 2.3: Decision boundaries of an unregularized tree (left) and a regularized tree (right)

```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))
```

```
0.898
0.92
```

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 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

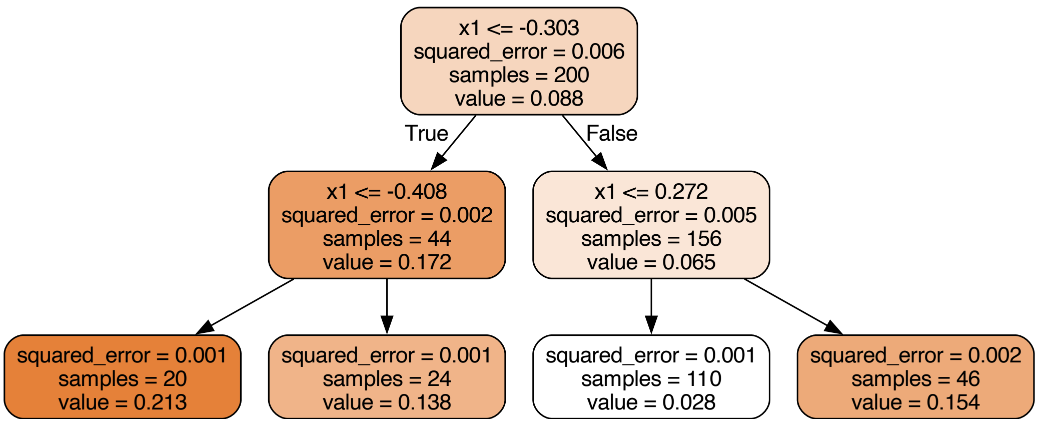

Figure 2.4: A DT for regression

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 works as described earlier, except that instead

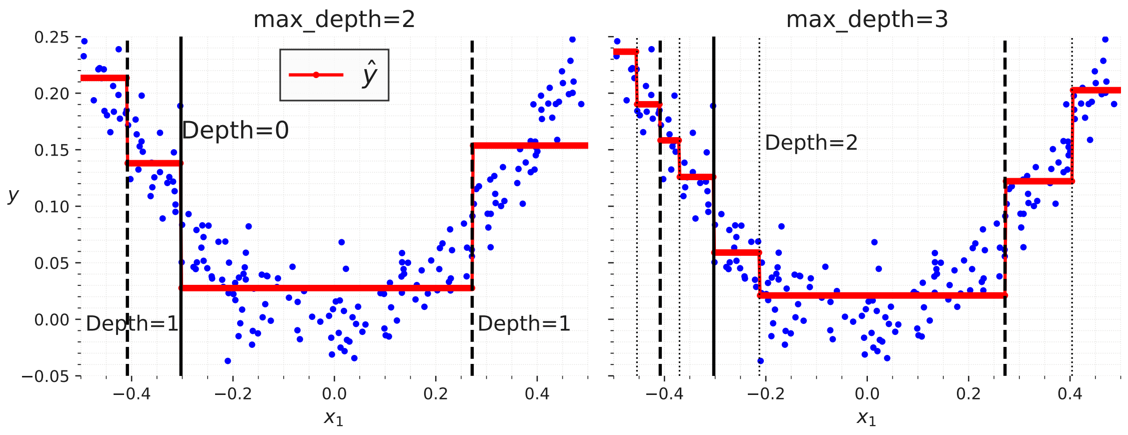

Figure 2.5: Predictions of two DT regression models

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.

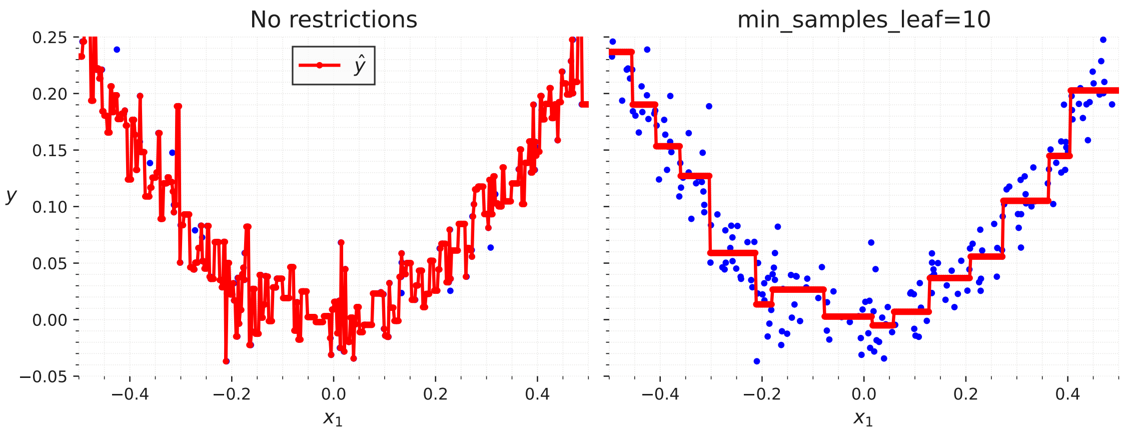

Figure 2.6: Predictions of an unregularized regression tree (left) and a regularized tree (right)

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

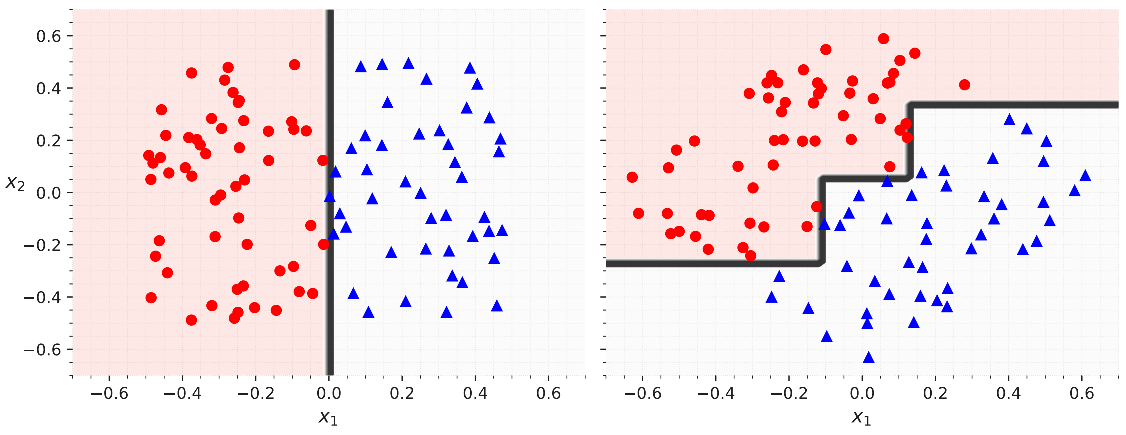

Figure 2.7: Sensitivity to training set rotation

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

```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.

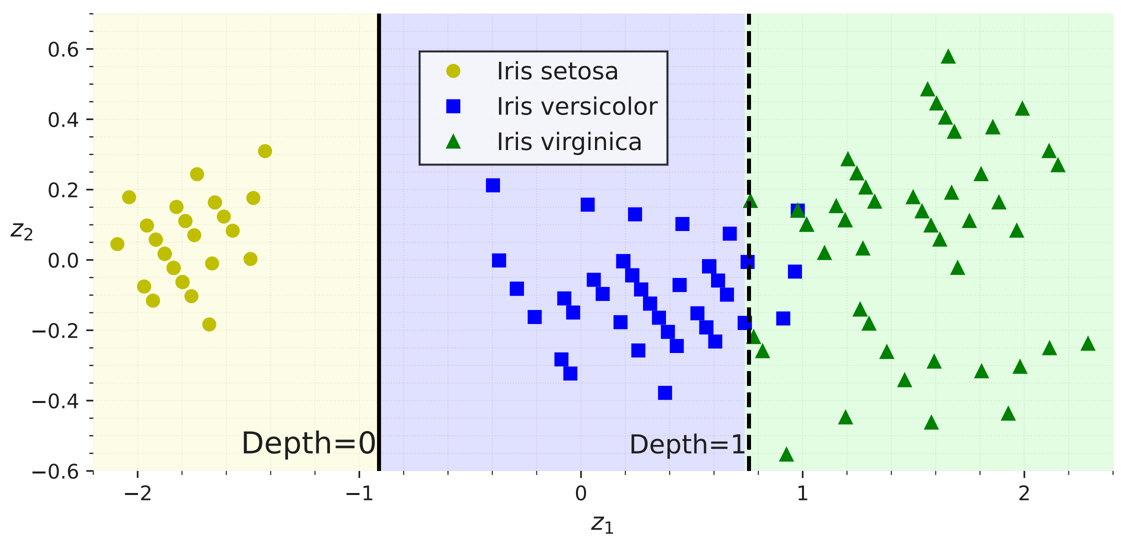

Figure 2.8: A tree's decision boundaries on the scaled and PCA-rotated iris dataset

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 over many trees, it's

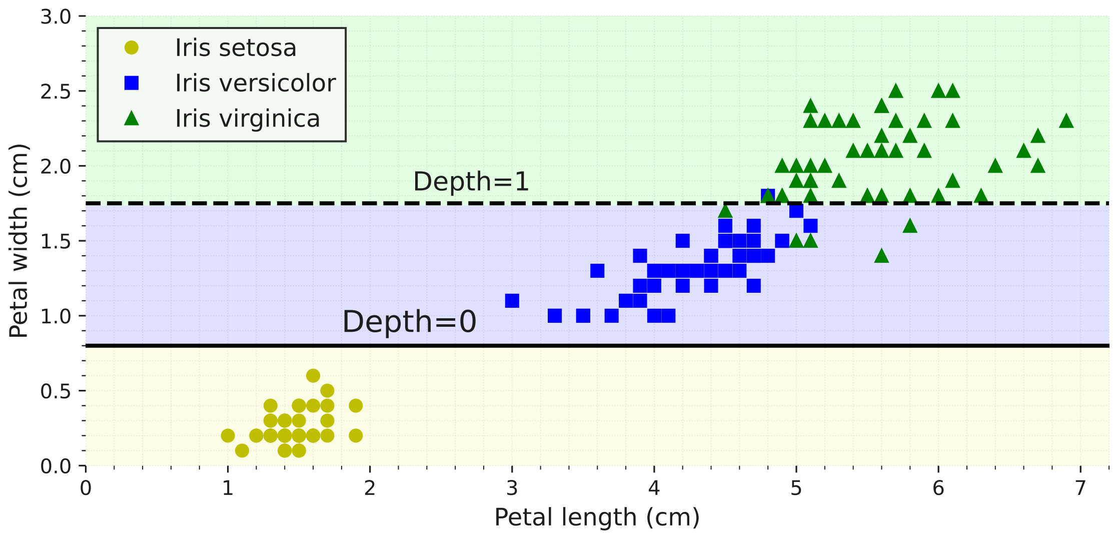

Figure 2.9: Retraining the same model on the same data may produce a very different model

possible to reduce variance significantly. 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

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.

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).

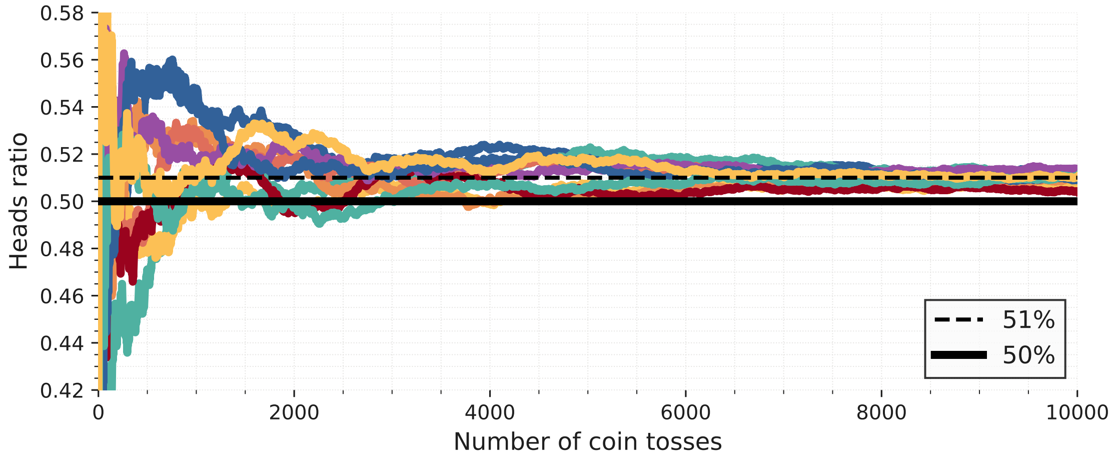

Figure 3.1: An example of a biased coin, where as the number of coin tosses increases the value of the will reach to 51%.

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:

```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
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:

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

```
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:

```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))
```

```
[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 hyperparameter 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:

```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))
```

```
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:

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

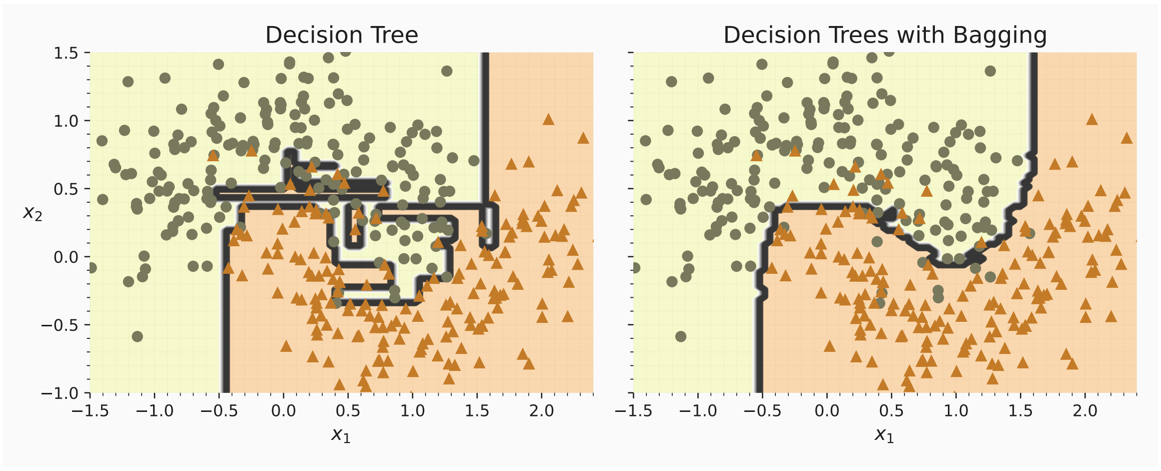

Figure 3.2: A single decision tree (left) versus a bagging ensemble of 500 trees (right)

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:

```
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_)
```

```
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:

```
from sklearn.metrics import accuracy_score

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

```
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:

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

```
[[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:

```python
from sklearn.ensemble import RandomForestClassifier

```

```python
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:

```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.

```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)
```

```
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

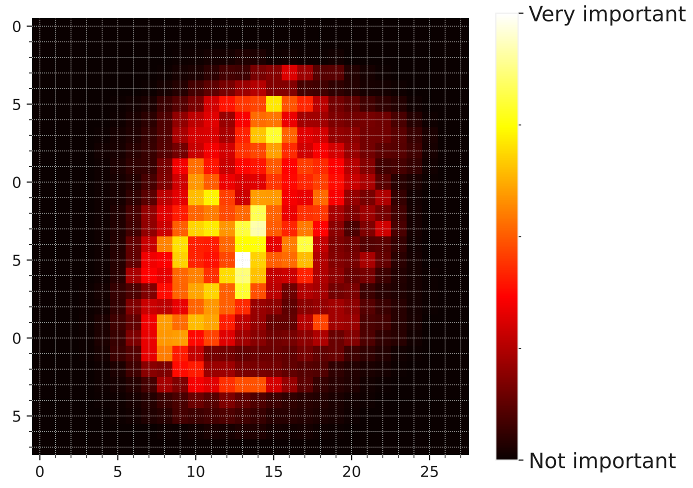

Figure 3.3: MNIST pixel importance (according to a RF classifier)

understanding of what features 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. 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 that 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 (such as 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 consecutive predictors on the moons dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF

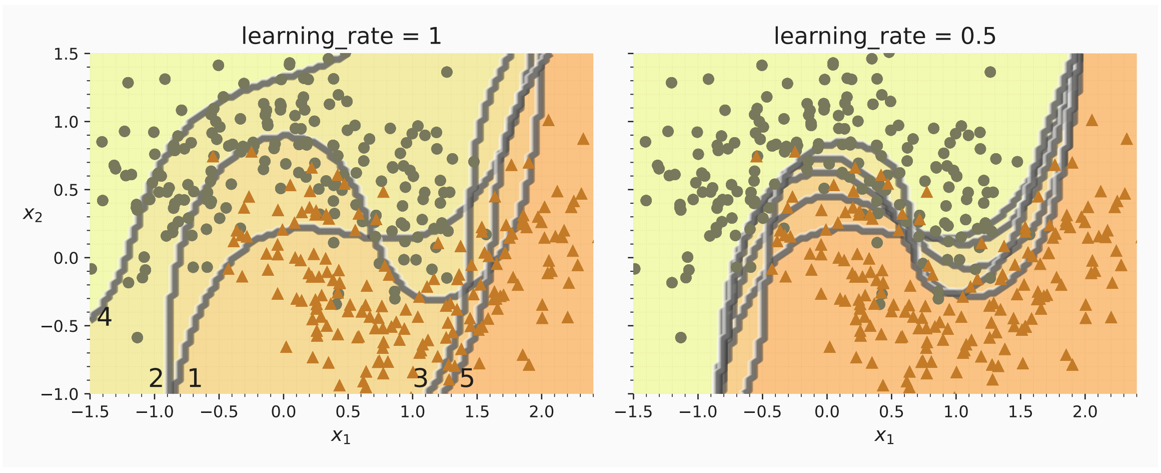

Figure 3.4: Decision boundaries of consecutive predictors.

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. As a result, 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.

$$r_{\mathrm{j}} = \frac{\sum_{i=1}}{\sum_{i=1}^{m} w^{(\mathrm{i})}}$$

where $y^{\mathrm{j(i)}}$ is the jth predictor's prediction for the ith instance. The predictor's weight α_j is then computed using Equation 7-2, where is the learning rate hyperparameter (this 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.

$$\alpha_j = \eta \log \frac{1 - r_\mathrm{j}}{r_\mathrm{j}}$$

Next, the AdaBoost algorithm updates the instance weights, using Equation 7-3, which boosts the weights of the misclassified instances. Equation 7-3. Weight update rule Then all the instance weights are normalized (i.e., divided by $\sum_{i=1}^{m} w^{(\mathrm{i})}$). 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 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—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:

```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)
```

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

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.

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:

```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:

```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:

```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 trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```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)))
```

```
[0.49484029 0.04021166 0.75026781]
```

Figure 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.

You 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:

```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)
```

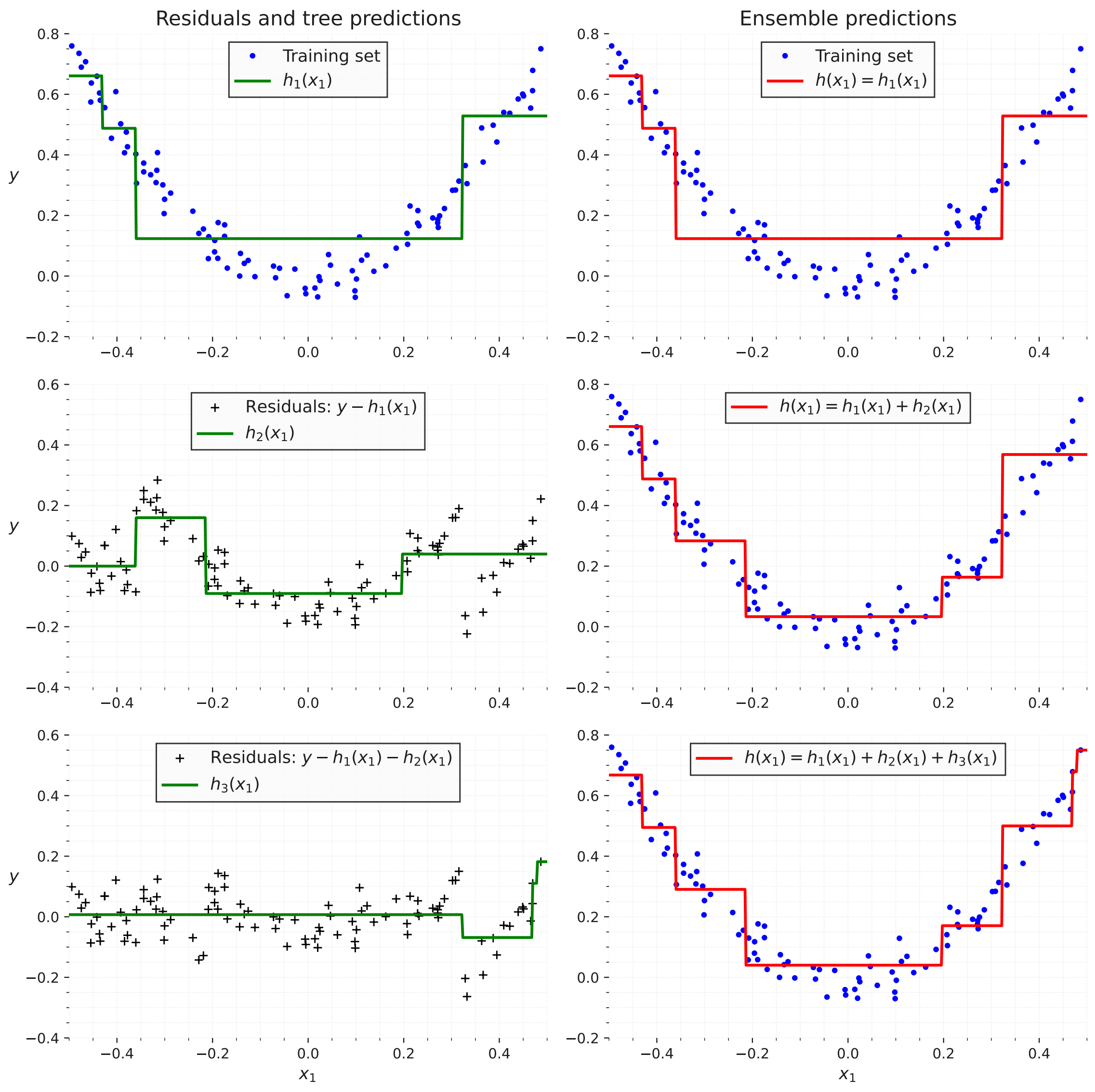

Figure 3.5: In this depiction of gradient boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

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. Figure 7-10 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-

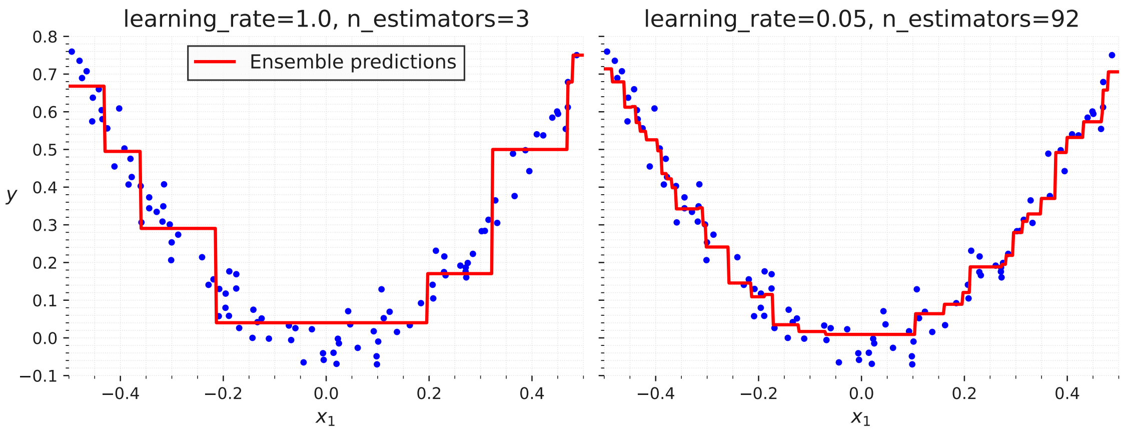

Figure 3.6: GBRT ensembles with not enough predictors (left) and just enough (right).

validation using GridSearchCV or RandomizedSearchCV, as usual, but there's a simpler 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:

```
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:

```
print(gbrt_best.n_estimators_)
```

```
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` also 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 O(b×m) instead of O(n×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 classes for HGB: HistGradientBoostingRegressor and 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 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 introduced in Chapter 2:

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

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:

```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.

Glossary

DT Decision Tree. 3, 4, 19, 21–31

LLN Law of Large Numbers. 3, 33

ML Machine Learning. 3, 6, 11, 12, 14, 19, 23, 25, 32, 52

OoB Out-of-Bag. 3, 4, 38, 39

RF Random Forest. 3, 32, 33, 40–43, 51, 52

SVM Support Vector Machines. 3, 4, 6, 7, 9, 10, 12, 13, 15–18, 34