O'REILLY®

Introduction to Machine Learning with Python

A GUIDE FOR DATA SCIENTISTS

Andreas C. Müller & Sarah Guido

Introduction to Machine Learning with Python
by Andreas C. Mueller and Sarah Guido

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Meghan Blanchette and Rachel Roumeliotis
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2016: First Edition

Revision History for the First Edition
2016-06-09: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491917213* for release details.

978-1-491-91721-3

[FILL IN]

Machine Learning with Python

Andreas C. Mueller and Sarah Guido

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Table of Contents

CHAPTER 1

Introduction

Machine learning is about extracting knowledge from data. It is a research field at the intersection of statistics, artificial intelligence and computer science, which is also known as predictive analytics or statistical learning. The application of machine learning methods has in recent years become ubiquitous in everyday life. From automatic recommendations of which movies to watch, to what food to order or which products to buy, to personalized online radio and recognizing your friends in your photos, many modern websites and devices have machine learning algorithms at their core.

When you look at at complex websites like Facebook, Amazon or Netflix, it is very likely that every part of the website you are looking at contains multiple machine learning models.

Outside of commercial applications, machine learning has had a tremendous influence on the way data driven research is done today. The tools introduced in this book have been applied to diverse scientific questions such as understanding stars, finding distant planets, analyzing DNA sequences, and providing personalized cancer treatments.

Your application doesn't need to be as large-scale or world-changing as these examples in order to benefit from machine learning. In this chapter, we will explain why machine learning became so popular, and dicuss what kind of problem can be solved using machine learning. Then, we will show you how to build your first machine learning model, introducing important concepts on the way.

Why machine learning?

In the early days of "intelligent" applications, many systems used hand-coded rules of "if" and "else" decisions to process data or adjust to user input. Think of a spam filter

whose job is to move an email to a spam folder. You could make up a black-list of words that would result in an email marked as spam. This would be an example of using an expert designed rule system to design an "intelligent" application. Designing kind of manual design of decision rules is feasible for some applications, in particular for those applications in which humans have a good understanding of how a decision should be made. However, using hand-coded rules to make decisions has two major disadvantages:

1. The logic required to make a decision is specific to a single domain and task. Changing the task even slightly might require a rewrite of the whole system.
2. Designing rules requires a deep understanding of how a decision should be made by a human expert.

One example of where this hand-coded approach will fail is in detecting faces in images. Today every smart phone can detect a face in an image. However, face detection was an unsolved problem until as recent as 2001. The main problem is that the way in which pixels (which make up an image in a computer) are "perceived by" the computer is very different from how humans perceive a face. This difference in representation makes it basically impossible for a human to come up with a good set of rules to describe what constitutes a face in a digital image.

Using machine learning, however, simply presenting a program with a large collection of images of faces is enough for an algorithm to determine what characteristics are needed to identify a face.

Problems that machine learning can solve

The most successful kind of machine learning algorithms are those that automate a decision making processes by generalizing from known examples. In this setting, which is known as a *supervised learning* setting, the user provides the algorithm with pairs of inputs and desired outputs, and the algorithm finds a way to produce the desired output given an input.

In particular, the algorithm is able to create an output for an input it has never seen before without any help from a human.

Going back to our example of spam classification, using machine learning, the user provides the algorithm a large number of emails (which are the input), together with the information about whether any of these emails are spam (which is the desired output). Given a new email, the algorithm will then produce a prediction as to whether or not the new email is spam.

Machine learning algorithms that learn from input-output pairs are called supervised learning algorithms because a "teacher" provides supervision to the algorithm in the form of the desired outputs for each example that they learn from.

While creating a dataset of inputs and outputs is often a laborious manual process, supervised learning algorithms are well-understood and their performance is easy to measure. If your application can be formulated as a supervised learning problem, and you are able to create a dataset that includes the desired outcome, machine learning will likely be able to solve your problem.

Examples of supervised machine learning tasks include:

- **Identifying the ZIP code from handwritten digits on an envelope.** Here the input is a scan of the handwriting, and the desired output is the actual digits in the zip code. To create a data set for building a machine learning model, you need to collect many envelopes. Then you can read the zip codes yourself and store the digits as your desired outcomes.

- **Determining whether or not a tumor is benign based on a medical image.** Here the input is the image, and the output is whether or not the tumor is benign. To create a data set for building a model, you need a database of medical images. You also need an expert opinion, so a doctor needs to look at all of the images and decide which tumors are benign and which are not.

- **Detecting fraudulent activity in credit card transactions.** Here the input is a record of the credit card transaction, and the output is whether it is likely to be fraudulent or not. Assuming that you are the entity distributing the credit cards, collecting a dataset means storing all transactions, and recording if a user reports any transaction as fraudulent.

An interesting thing to note about the three examples above is that although the inputs and outputs look fairly straight-forward, the data collection process for these three tasks is vastly different.

While reading envelopes is laborious, it is easy and cheap. Obtaining medical imaging and expert opinions on the other hand not only requires expensive machinery but also rare and expensive expert knowledge, not to mention ethical concerns and privacy issues. In the example of detecting credit card fraud, data collection is much simpler. Your customers will provide you with the desired output, as they will report fraud. All you have to do to obtain the input output pairs of fraudulent and non-fraudulent activity is wait.

The other type of algorithms that we will cover in this book is unsupervised algorithms. In unsupervised learning, only the input data is known and there is no known output data given to the algorithm. While there are many successful applications of these methods as well, they are usually harder to understand and evaluate.

Examples of unsupervised learning include:

- **Identifying topics in a set of blog posts.** If you have a large collection of text data, you might want to summarize it and find prevalent themes in it. You might not know beforehand what these topics are, or how many topics there might be. Therefore, there are no known outputs.

- **Segmenting customers into groups with similar preferences.** Given a set of customer records, you might want to identify which customers are similar, and whether there are groups of customers with similar preferences. For a shopping site these might be "parents", "bookworms" or "gamers". Since you don't know in advanced what these groups might be, or even how many there are, you have no known outputs.

- **Detecting abnormal access patterns to a website.** To identify abuse or bugs, it is often helpful to find access patterns that are different from the norm. Each abnormal pattern might be very different, and you might not have any recorded instances of abnormal behavior. Since in this example you only observe traffic, and you don't know what constitutes normal and abnormal behavior, this is an unsupervised problem.

For both supervised and unsupervised learning tasks, it is important to have a representation of your input data that a computer can understand. Often it is helpful to think of your data as a table. Each data point that you want to reason about (each email, each customer, each transaction) is a row, and each property that describes that data point (say the age of a customer, the amount or location of a transaction) is a column.

You might describe users by their age, their gender, when they created an account and how often they bought from your online shop. You might describe the image of a tumor by the gray-scale values of each pixel, or maybe by using the size, shape and color of the tumor to describe it.

Each entity or row here is known as data point or *sample* in machine learning, while the columns, the properties that describe these entities, are called *features*.

We will later go into more detail on the topic of building a good representation of your data, which is called feature extraction or feature engineering. You should keep in mind however that no machine learning algorithm will be able to make a prediction on data for which it has no information. For example, if the only feature that you have for a patient is their last name, no algorithm will be able to predict their gender. This information is simply not contained in your data. If you add another feature that contains their first name, you will have much better luck, as it is often possible to tell the gender by a person's first name.

Knowing your data

Quite possibly the most important part in the machine learning process is understanding the data you are working with. It will not be effective to randomly choose an algorithm and throw your data at it. It is necessary to understand what is going on in your dataset before you begin building a model. Each algorithm is different in terms of what data it works best for, what kinds data it can handle, what kind of data it is optimized for, and so on. Before you start building a model, it is important to know the answers to most of, if not all of, the following questions:

- How much data do I have? Do I need more?
- How many features do I have? Do I have too many? Do I have too few?
- Is there missing data? Should I discard the rows with missing data or handle them differently?
- What question(s) am I trying to answer? Do I think the data collected can answer that question?

The last bullet point is the most important question, and certainly is not easy to answer. Thinking about these questions will help drive your analysis.

Keeping these basics in mind as we move through the book will prove helpful, because while scikit-learn is a fairly easy tool to use, it is geared more towards those with domain knowledge in machine learning.

Why Python?

Python has become the lingua franca for many data science applications. It combines the powers of general purpose programming languages with the ease of use of domain specific scripting languages like matlab or R.

Python has libraries for data loading, visualization, statistics, natural language processing, image processing, and more. This vast toolbox provides data scientists with a large array of general and special purpose functionality.

As a general purpose programming language, Python also allows for the creation of complex graphic user interfaces (GUIs), web services and for integration into existing systems.

What this book will cover

In this book, we will focus on applying machine learning algorithms for the purpose of solving practical problems. We will focus on how to write applications using the machine learning library scikit-learn for the Python programming language. Impor-

tant aspects that we will cover include formulating tasks as machine learning problems, preprocessing data for use in machine learning algorithms, and choosing appropriate algorithms and algorithmic parameters.

We will focus mostly on supervised learning techniques and algorithms, as these are often the most useful ones in practice, and they are easy for beginners to use and understand.

We will also discuss several common types of input, including text data.

What this book will not cover

This book will not cover the mathematical details of machine learning algorithms, and we will keep the number of formulas that we include to a minimum. In particular, we will not assume any familiarity with linear algebra or probability theory. As mathematics, in particular probability theory, is the foundation upon which machine learning is build, we will not be able to go into the analysis of the algorithms in great detail. If you are interested in the mathematics of machine learning algorithms, we recommend the text book "Elements of Statistical Learning" by Hastie, Tibshirani and Friedman, which is available for free at the authors website[footnote: http://statweb.stanford.edu/~tibs/ElemStatLearn/]. We will also not describe how to write machine learning algorithms from scratch, and will instead focus on how to use the large array of models already implemented in scikit-learn and other libraries.

We will not discuss reinforcement learning, which is about an agent learning from its interaction with an environment, and we will only briefly touch upon deep learning.

Some of the algorithms that are implemented in scikit-learn but are outside the scope of this book include Gaussian Processes, which are complex probabilistic models, and semi-supervised models, which work with supervised information on only some of the samples.

We will not also explicitly talk about how to work with time-series data, although many of techniques we discuss are applicable to this kind of data as well. Finally, we will not discuss how to do machine learning on natural images, as this is beyond the scope of this book.

Scikit-learn

Scikit-learn is an open-source project, meaning that scikit-learn is free to use and distribute, and anyone can easily obtain the source code to see what is going on behind the scenes. The scikit-learn project is constantly being developed and improved, and has a very active user community. It contains a number of state-of-the-art machine learning algorithms, as well as comprehensive documentation about each algorithm on the website [footnote http://scikit-learn.org/stable/documentation]. Scikit-learn is

a very popular tool, and the most prominent Python library for machine learning. It is widely used in industry and academia, and there is a wealth of tutorials and code snippets about scikit-learn available online. Scikit-learn works well with a number of other scientific Python tools, which we will discuss later in this chapter.

While studying the book, we recommend that you also browse the scikit-learn user guide and API documentation for additional details, and many more options to each algorithm. The online documentation is very thorough, and this book will provide you with all the prerequisites in machine learning to understand it in detail.

Installing Scikit-learn

Scikit-learn depends on two other Python packages, NumPy and SciPy. For plotting and interactive development, you should also install matplotlib, IPython and the Jupyter notebook. We recommend using one of the following pre-packaged Python distributions, which will provide the necessary packages:

- Anaconda (*https://store.continuum.io/cshop/anaconda/*): a Python distribution made for large-scale data processing, predictive analytics, and scientific computing. Anaconda comes with NumPy, SciPy, matplotlib, IPython, Jupyter notebooks, and scikit-learn. Anaconda is available on Mac OS X, Windows, and Linux.

- Enthought Canopy (*https://www.enthought.com/products/canopy/*): another Python distribution for scientific computing. This comes with NumPy, SciPy, matplotlib, and IPython, but the free version does not come with scikit-learn. If you are part of an academic, degree-granting institution, you can request an academic license and get free access to the paid subscription version of Enthought Canopy. Enthought Canopy is available for Python 2.7.x, and works on Mac, Windows, and Linux.

- Python(x,y) (*https://code.google.com/p/pythonxy/*): a free Python distribution for scientific computing, specifically for Windows. Python(x,y) comes with NumPy, SciPy, matplotlib, IPython, and scikit-learn.

If you already have a python installation set up, you can use pip to install any of these packages.

```
$ pip install numpy scipy matplotlib ipython scikit-learn
```

We do not recommended using pip to install NumPy and SciPy on Linux, as it involves compiling the packages from source. See the scikit-learn website for more detailed installation.

Essential Libraries and Tools

Understanding what scikit-learn is and how to use it is important, but there are a few other libraries that will enhance your experience. Scikit-learn is built on top of the NumPy and SciPy scientific Python libraries. In addition to knowing about NumPy and SciPy, we will be using Pandas and matplotlib. We will also introduce the Jupyter Notebook, which is an browser-based interactive programming environment. Briefly, here is what you should know about these tools in order to get the most out of scikit-learn.

If you are unfamiliar with numpy or matplotlib, we recommend reading the first chapter of the scipy lecture notes[footnote: http://www.scipy-lectures.org/].

Jupyter Notebook

The Jupyter Notebook is an interactive environment for running code in the browser. It is a great tool for exploratory data analysis and is widely used by data scientists. While Jupyter Notebook supports many programming languages, we only need the Python support. The Jypyter Notebook makes it easy to incorporate code, text, and images, and all of this book was in fact written as an IPython notebook.

All of the code examples we include can be downloaded from github [FIXME add github footnote].

NumPy

NumPy is one of the fundamental packages for scientific computing in Python. It contains functionality for multidimensional arrays, high-level mathematical functions such as linear algebra operations and the Fourier transform, and pseudo random number generators.

The NumPy array is the fundamental data structure in scikit-learn. Scikit-learn takes in data in the form of NumPy arrays. Any data you're using will have to be converted to a NumPy array. The core functionality of NumPy is this "ndarray", meaning it has *n* dimensions, and all elements of the array must be the same type. A NumPy array looks like this:

```
import numpy as np

x = np.array([[1, 2, 3], [4, 5, 6]])
x

array([[1, 2, 3],

       [4, 5, 6]])
```

SciPy

SciPy is both a collection of functions for scientific computing in python. It provides, among other functionality, advanced linear algebra routines, mathematical function optimization, signal processing, special mathematical functions and statistical distributions. Scikit-learn draws from SciPy's collection of functions for implementing its algorithms.

The most important part of scipy for us is `scipy.sparse` with provides *sparse matrices*, which is another representation that is used for data in scikit-learn. Sparse matrices are used whenever we want to store a 2d array that contains mostly zeros:

```
from scipy import sparse

# create a 2d numpy array with a diagonal of ones, and zeros everywhere else
eye = np.eye(4)
print("Numpy array:\n%s" % eye)

# convert the numpy array to a scipy sparse matrix in CSR format
# only the non-zero entries are stored
sparse_matrix = sparse.csr_matrix(eye)
print("\nScipy sparse CSR matrix:\n%s" % sparse_matrix)
```

```
Numpy array:

[[ 1.  0.  0.  0.]

 [ 0.  1.  0.  0.]

 [ 0.  0.  1.  0.]

 [ 0.  0.  0.  1.]]



Scipy sparse CSR matrix:

  (0, 0)    1.0

  (1, 1)    1.0

  (2, 2)    1.0

  (3, 3)    1.0
```

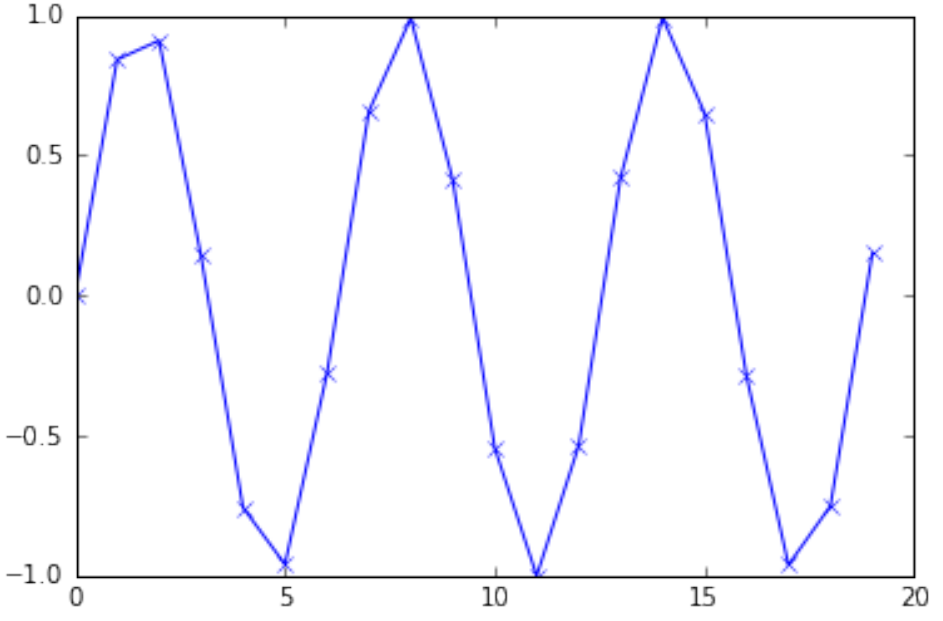
More details on scipy sparse matrices can be found in the scipy lecture notes.

matplotlib

Matplotlib is the primary scientific plotting library in Python. It provides function for making publication-quality visualizations such as line charts, histograms, scatter

plots, and so on. Visualizing your data and any aspects of your analysis can give you important insights, and we will be using matplotlib for all our visualizations.

```
%matplotlib inline
import matplotlib.pyplot as plt

# Generate a sequence of integers
x = np.arange(20)
# create a second array using sinus
y = np.sin(x)
# The plot function makes a line chart of one array against another
plt.plot(x, y, marker="x")
```



Pandas

Pandas is a Python library for data wrangling and analysis. It is built around a data structure called DataFrame, that is modeled after the R DataFrame. Simply put, a Pandas Pandas DataFrame is a table, similar to an Excel Spreadsheet. Pandas provides a great range of methods to modify and operate on this table, in particular it allows SQL-like queries and joins of tables. Another valuable tool provided by Pandas is its ability to ingest from a great variety of file formats and databases, like SQL, Excel files and comma separated value (CSV) files. Going into details about the functionality of Pandas is out of the scope of this book. However, "Python for Data Analysis" by Wes McKinney provides a great guide.

Here is a small example of creating a DataFrame using a dictionary:

```
import pandas as pd

# create a simple dataset of people
data = {'Name': ["John", "Anna", "Peter", "Linda"],
        'Location' : ["New York", "Paris", "Berlin", "London"],
        'Age' : [24, 13, 53, 33]
       }

data_pandas = pd.DataFrame(data)
data_pandas
```

	Age	Location	Name
0	24	New York	John
1	13	Paris	Anna
2	53	Berlin	Peter
3	33	London	Linda

Python2 versus Python3

There are two major versions of Python that are widely used at the moment: Python2 (more precisely 2.7) and Python3 (with the latest release being 3.5 at the time of writing), which sometimes leads to some confusion. Python2 is no longer actively developed, but because Python3 contains major changes, Python2 code does usually not run without changes on Python3. If you are new to Python, or are starting a new project from scratch, we highly recommend using the latests version of Python3.

If you have a large code-base that you rely on that is written for Python2, you are excused from upgrading for now. However, you should try to migrate to Python3 as soon as possible. Writing any new code, it is for the most part quite easy to write code that runs under Python2 and Python3 [Footnote: The `six` package can be very handy for that].

All the code in this book is written in a way that works for both versions. However, the exact output might differ slightly under Python2.

Versions Used in this Book

We are using the following versions of the above libraries in this book:

```
import pandas as pd
print("pandas version: %s" % pd.__version__)

import matplotlib
print("matplotlib version: %s" % matplotlib.__version__)

import numpy as np
print("numpy version: %s" % np.__version__)
```

```
import IPython
print("IPython version: %s" % IPython.__version__)

import sklearn
print("scikit-learn version: %s" % sklearn.__version__)
```

```
pandas version: 0.17.1

matplotlib version: 1.5.1

numpy version: 1.10.4

IPython version: 4.1.2

scikit-learn version: 0.18.dev0
```

While it is not important to match these versions exactly, you should have a version of scikit-learn that is as least as recent as the one we used.

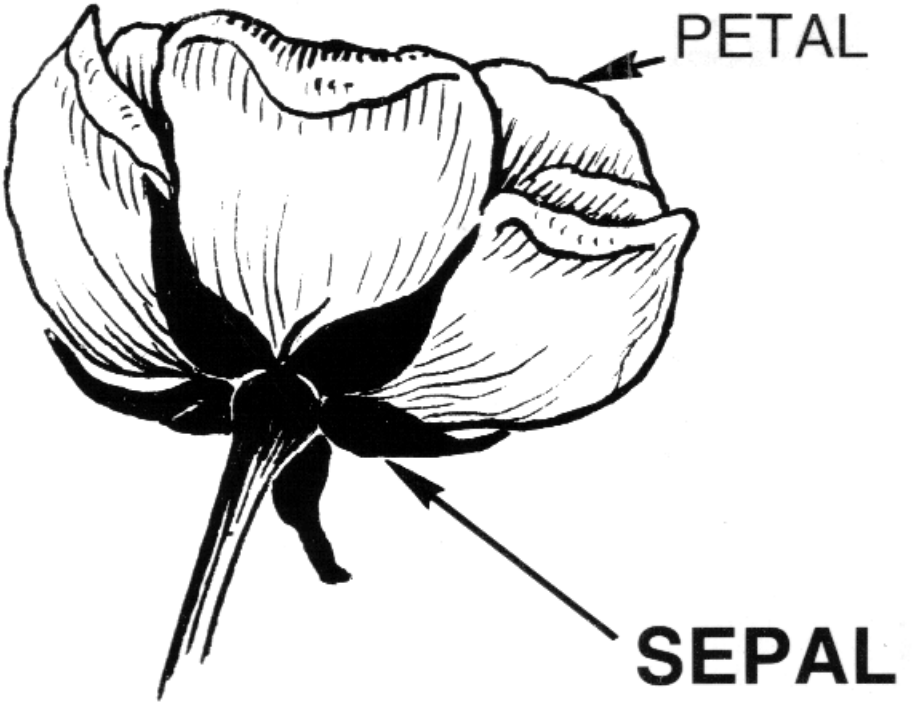
Now that we have everything set up, let's dive into our first appication of machine learning.

A First Application: Classifying iris species

In this section, we will go through a simple machine learning application and create our first model.

In the process, we will introduce some core concepts and nomenclature for machine learning.

Let's assume that a hobby botanist is interested in distinguishing what the species is of some iris flowers that she found. She has collected some measurements associated with the iris: the length and width of the petals, and the length and width of the sepal, all measured in centimeters.



She also has the measurements of some irises that have been previously identified by an expert botanist as belonging to the species Setosa, Versicolor or Virginica. For these measurements, she can be certain of which species each iris belongs to. Let's assume that these are the only species our hobby botanist will encounter in the wild.

Our goal is to build a machine learning model that can learn from the measurements of these irises whose species is known, so that we can predict the species for a new iris.

Since we have measurements for which we know the correct species of iris, this is a supervised learning problem. In this problem, we want to predict one of several options (the species of iris). This is an example of a *classification* problem. The possible outputs (different species of irises) are called *classes*.

Since every iris in the dataset belongs to one of three classes this problem is a three-class classification problem.

The desired output for a single data point (an iris) is the species of this flower. For a particular data point, the species it belongs to is called its *label*.

Meet the data

The data we will use for this example is the iris dataset, a classical dataset in machine learning an statistics.

It is included in scikit-learn in the dataset module. We can load it by calling the `load_iris` function:

```
from sklearn.datasets import load_iris
iris = load_iris()
```

The iris object that is returned by `load_iris` is a *Bunch* object, which is very similar to a dictionary. It contains keys and values:

```
iris.keys()
```

```
dict_keys(['DESCR', 'data', 'target_names', 'feature_names', 'target'])
```

The value to the key `DESCR` is a short description of the dataset. We show the beginning of the description here. Feel free to look up the rest yourself.

```
print(iris['DESCR'][:193] + "\n...")
```

```
Iris Plants Database

====================



Notes

-----

Data Set Characteristics:

    :Number of Instances: 150 (50 in each of three classes)

    :Number of Attributes: 4 numeric, predictive att

...
```

The value with key `target_names` is an array of strings, containing the species of flower that we want to predict:

```
iris['target_names']
```

```
array(['setosa', 'versicolor', 'virginica'],

      dtype='<U10')
```

The `feature_names` are a list of strings, giving the description of each feature:

```
iris['feature_names']
```

```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

The data itself is contained in the `target` and `data` fields. The `data` contains the numeric measurements of sepal length, sepal width, petal length, and petal width in a numpy array:

```
type(iris['data'])
numpy.ndarray
```

The rows in the data array correspond to flowers, while the columns represent the four measurements that were taken for each flower:

```
iris['data'].shape
(150, 4)
```

We see that the data contains measurements for 150 different flowers.

Remember that the individual items are called *samples* in machine learning, and their properties are called *features*.

The shape of the data array is the number of samples times the number of features.

This is a convention in scikit-learn, and your data will always be assumed to be in this shape.

Here are the feature values for the first five samples:

```
iris['data'][:5]
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2]])
```

The `target` array contains the species of each of the flowers that were measured, also as a numpy array:

```
type(iris['target'])
numpy.ndarray
```

The target is a one-dimensional array, with one entry per flower:

```
iris['target'].shape
```

```
(150,)
```

The species are encoded as integers from 0 to 2:

```
iris['target']
```

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

The meaning of the numbers are given by the `iris['target_names']` array: 0 means Setosa, 1 means Versicolor and 2 means Virginica.

Measuring Success: Training and testing data

We want to build a machine learning model from this data that can predict the species of iris for a new set of measurements.

Before we can apply our model to new measurements, we need to know whether our model actually works, that is whether we should trust its predictions.

Unfortunately, we can not use the data we use to build the model to evaluate it. This is because our model can always simply remember the whole training set, and will therefore always predict the correct label for any point in the training set. This "remembering" does not indicate to us whether our model will *generalize* well, in other words whether it will also perform well on new data. So before we apply our model to new measurements, we will want to know whether we can trust its predictions.

To assess the models' performance, we show the model new data (that it hasn't seen before) for which we have labels. This is usually done by splitting the labeled data we have collected (here our 150 flower measurements) into two parts.

The part of the data is used to build our machine learning model, and is called the *training data* or *training set*. The rest of the data will be used to access how well the model works and is called *test data*, *test set* or *hold-out set*.

Scikit-learn contains a function that shuffles the dataset and splits it for you, the `train_test_split` function.

This function extracts 75% of the rows in the data as the training set, together with the corresponding labels for this data. The remaining 25% of the data, together with the remaining labels are declared as the test set.

How much data you want to put into the training and the test set respectively is somewhat arbitrary, but using a test-set containing 25% of the data is a good rule of thumb.

In scikit-learn, data is usually denoted with a capital `X`, while labels are denoted by a lower-case `y`.

Let's call `train_test_split` on our data and assign the outputs using this nomenclature:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(iris['data'], iris['target'],
                                                    random_state=0)
```

The `train_test_split` function shuffles the dataset using a pseudo random number generator before making the split. If we would take the last 25% of the data as a test set, all the data point would have the label `2`, as the data points are sorted by the label (see the output for `iris['target']` above). Using a tests set containing only one of the three classes would not tell us much about how well we generalize, so we shuffle our data, to make sure the test data contains data from all classes.

To make sure that we will get the same output if we run the same function several times, we provide the pseudo random number generator with a fixed seed using the `random_state` parameter. This will make the outcome deterministic, so this line will always have the same outcome. We will always fix the `random_state` in this way when using randomized procedures in this book.

The output of the `train_test_split` function are `X_train`, `X_test`, `y_train` and `y_test`, which are all numpy arrays. `X_train` contains 75% of the rows of the dataset, and `X_test` contains the remaining 25%:

```
X_train.shape
```

```
(112, 4)
```

```
X_test.shape
```

```
(38, 4)
```

First things first: Look at your data

Before building a machine learning model, it is often a good idea to inspect the data, to see if the task is easily solvable without machine learning, or if the desired information might not be contained in the data.

Additionally, inspecting your data is a good way to find abnormalities and peculiarities. Maybe some of your irises were measured using inches and not centimeters, for example. In the real world, inconsistencies in the data and unexpected measurements are very common.

One of the best ways to inspect data is to visualize it. One way to do this is by using a scatter plot.

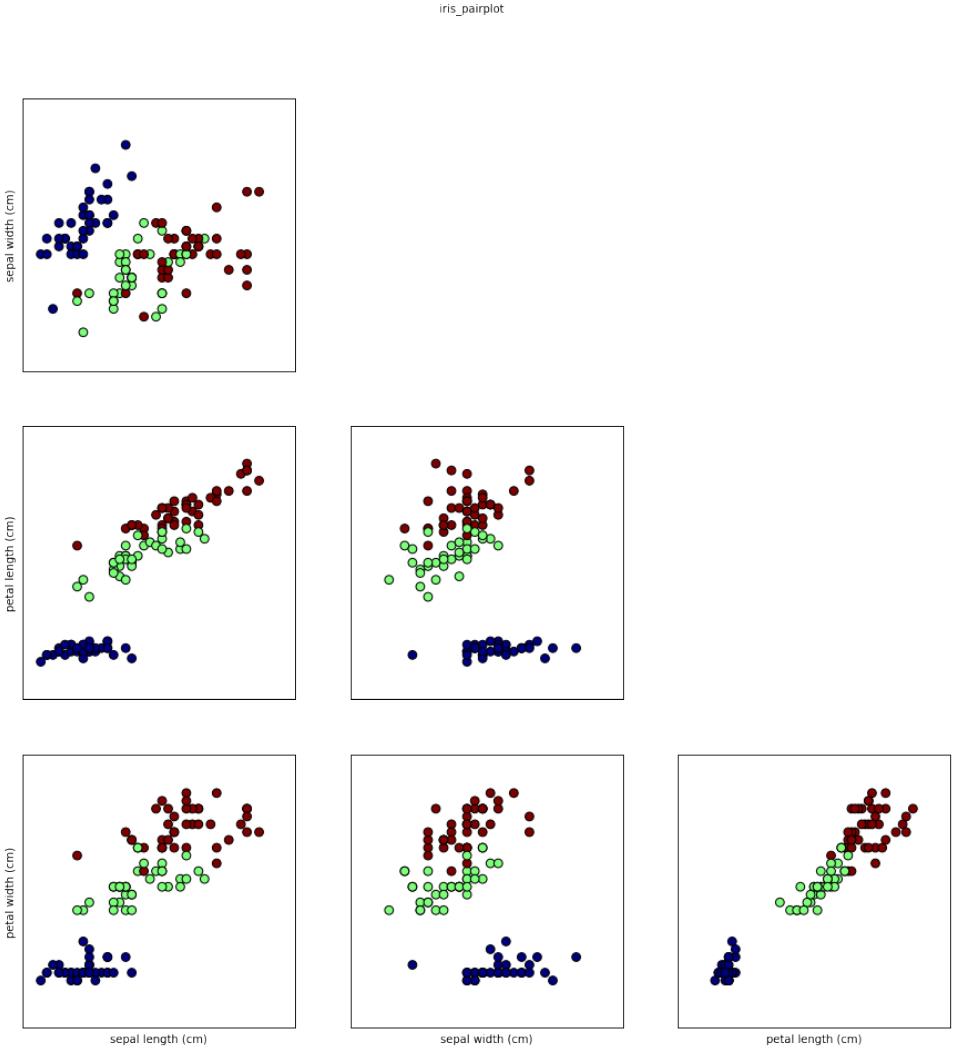
A scatter plot of the data puts one feature along the x-axis, one feature along the y-axis, and draws a dot for each data point.

Unfortunately, computer screens have only two dimensions, which allows us to only plot two (or maybe three) features at a time. It is difficult to plot datasets with more than three features this way.

One way around this problem is to do a pair plot, which looks at all pairs of two features. If you have a small number of features, such as the four we have here, this is quite reasonable. You should keep in mind that a pair plot does not show the interaction of all of features at once, so some interesting aspects of the data may not be revealed when visualizing it this way.

Here is a pair plot of the features in the training set. The data points are colored according to the species the iris belongs to:

```
fig, ax = plt.subplots(3, 3, figsize=(15, 15))
plt.suptitle("iris_pairplot")

for i in range(3):
    for j in range(3):
        ax[i, j].scatter(X_train[:, j], X_train[:, i + 1], c=y_train, s=60)
        ax[i, j].set_xticks(())
        ax[i, j].set_yticks(())
        if i == 2:
            ax[i, j].set_xlabel(iris['feature_names'][j])
        if j == 0:
            ax[i, j].set_ylabel(iris['feature_names'][i + 1])
        if j > i:
            ax[i, j].set_visible(False)
```



From the plots, we can see that the three classes seem to be relatively well separated using the sepal and petal measurements. This means that a machine learning model will likely be able to learn to separate them.

Building your first model: k nearest neighbors

Now we can start building the actual machine learning model. There are many classification algorithms in scikit-learn that we could use. Here we will use a k nearest neighbors classifier, which is easy to understand.

Building this model only consists of storing the training set. To make a prediction for a new data point, the algorithm finds the point in the training set that is closest to the new point. Then, it and assigns the label of this closest data training point to the new data point.

The `k` in `k` nearest neighbors stands for the fact that instead of using only the closest neighbor to the new data point, we can consider any fixed number `k` of neighbors in the training (for example, the closest three or five neighbors). Then, we can make a prediction using the majority class among these neighbors. We will go into more details about this later.

Let's use only a single neighbor for now.

All machine learning models in scikit-learn are implemented in their own class, which are called `Estimator` classes. The `k` nearest neighbors classification algorithm is implemented in the `KNeighborsClassifier` class in the `neighbors` module.

Before we can use the model, we need to instantiate the class into an object. This is when we will set any parameters of the model. The single parameter of the `KNeighborsClassifier` is the number of neighbors, which we will set to one:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

The `knn` object encapsulates the algorithm to build the model from the training data, as well the algorithm to make predictions on new data points.

It will also hold the information the algorithm has extracted from the training data. In the case of `KNeighborsClassifier`, it will just store the training set.

To build the model on the training set, we call the `fit` method of the `knn` object, which takes as arguments the numpy array `X_train` containing the training data and the numpy array `y_train` of the corresponding training labels:

```
knn.fit(X_train, y_train)

KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',

           metric_params=None, n_jobs=1, n_neighbors=1, p=2,

           weights='uniform')
```

Making predictions

We can now make predictions using this model on new data, for which we might not know the correct labels.

Imagine we found an iris in the wild with a sepal length of 5cm, a sepal width of 2.9cm, a petal length of 1cm and a petal width of 0.2cm. What species of iris would this be?

We can put this data into a numpy array, again with the shape number of samples (one) times number of features (four):

```
X_new = np.array([[5, 2.9, 1, 0.2]])
X_new.shape
```

```
(1, 4)
```

To make prediction we call the `predict` method of the `knn` object:

```
prediction = knn.predict(X_new)
prediction
```

```
array([0])
```

```
iris['target_names'][prediction]
```

```
array(['setosa'],

      dtype='<U10')
```

Our model predicts that this new iris belongs to the class 0, meaning its species is Setosa.

But how do we know whether we can trust our model? We don't know the correct species of this sample, which is the the whole point of building the model!

Evaluating the model

This is where the test set that we created earlier comes in. This data was not used to build the model, but we do know what the correct species are for each iris in the test set.

We can make a prediction for an iris in the test data, and compare it against its label (the known species). We can measure how well the model works by computing the *accuracy*, which is the fraction of flowers for which the right species was predicted:

```
y_pred = knn.predict(X_test)
np.mean(y_pred == y_test)
```

```
0.97368421052631582
```

We can also use the `score` method of the `knn` object, which will compute the test set accuracy for us:

```
knn.score(X_test, y_test)
```

```
0.97368421052631582
```

For this model, the test set accuracy is about 0.97, which means we made the right prediction for 97% of the irises in the test set. Under some mathematical assumptions, this means that we can expect our model to be correct 97% of the time for new irises.

For our hobby botanist application, this high level of accuracy means that our models may be trustworthy enough to use. In later chapters we will discuss how we can improve performance, and what caveats there are in tuning a model.

Summary

Let's summarize what we learned in this chapter. We started off formulating a task of predicting which species of iris a particular flower belongs to by using physical measurements of the flower. We used a dataset of measurements that was annotated by an expert with the correct species to build our model, making this a supervised learning task. There were three possible species, Setosa, Versicolor or Virginica, which made the task a three-class *classification* problem. The possible species are called *classes* in the classification problem, and the species of a single iris is called its *label*.

The dataset consists of two numpy arrays, one containing the data, which is referred to as `X` in scikit-learn, and one containing the correct or desired outputs, which is called `y`. The array `X` is a two-dimensional array of features, with one row per data point, and one column per feature. The array `y` is a one-dimensional array, which here contained one class label from 0 to 2 for each of the samples.

We split our dataset into a *training set*, to build our model, and a *test set*, to evaluate how well our model will generalize to new, unseen data.

We chose the `k` nearest neighbors classification algorithm, which makes predictions for a new data point by considering its closest neighbor(s) in the training set.

The algorithm is implemented in the `KNeighborsClassifier` class, which contains the algorithm to build the model, as well as the algorithm to make a prediction using the model. We instantiated the class, setting parameters. Then, we built the model by calling the `fit` method, passing the training data `X_train` and training outputs `y_train` as parameters.

We evaluated the model using the `score` method, that computes the *accuracy* of the model. We applied the `score` method to the test set data and the test set labels, and found that our model is about 97% accurate, meaning it is correct 97% of the time on the test set.

This gave us the confidence to apply the model to new data (in our example, new flower measurements), and trust that the model will be correct about 97% of the time.

Here is a summary of the code needed for the whole training and evaluation procedure:

```
X_train, X_test, y_train, y_test = train_test_split(iris['data'], iris['target'],
                                                    random_state=0)

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)

knn.score(X_test, y_test)
```

```
0.97368421052631582
```

This snippet contains the core code for applying any machine learning algorithms using scikit-learn. The `fit`, `predict` and `score` methods are the common interface to supervised models in scikit-learn, and with the concepts introduced in this chapter, you can apply these models to many machine learning tasks.

In the next chapter, we will go into more depth about the different kinds of supervised models in scikit-learn, and how to apply them successfully.

CHAPTER 2

Supervised Learning

As we mentioned in the introduction, supervised machine learning is one of the most commonly used and successful types of machine learning. In this chapter, we will describe supervised learning in more detail, and explain several popular supervised learning algorithms.

We already saw an application of supervised machine learning in the last chapter: classifying iris flowers into several species using physical measurements of the flowers.

Remember that supervised learning is used whenever we want to predict a certain outcome from a given input, and we have examples of input-output pairs. We build a machine learning model from these input-output pairs, which comprise our training set. Our goal is to make accurate predictions to new, never-before seen data.

Supervised learning often requires human effort to build the training set, but afterwards automates and often speeds up an otherwise laborious or infeasible task.

Classification and Regression

There are two major types of supervised machine learning algorithms, called *classification* and *regression*.

In classification, the goal is to predict a *class label*, which is a choice from a predefined list of possibilities. In Chapter 1 (Introduction) we used the example of classifying irises into one of three possible species. Classification is sometimes separated into *binary classification*, which is the special case of distinguishing between exactly two classes, and *multi-class classification* which is classificaFItion between more than two classes. You can think of binary classification as trying to answer a "yes" or "no" question.

Classifying emails into either spam or not spam is an example of a binary classification problem. In this binary classification task, the yes or no question being asked would be "Is this email spam?".

[info box] In binary classification we often speak of one class being the *positive* class and the other class being the *negative* class. Here, positive don't represent benefit or value, but rather what the object of study is. So when looking for spam, "positive" could mean the spam class. Which of the two classes is called positive is often a subjective manner, and specific to the domain.FI

[/info box]

The iris example on the other hand is an example of a multi-class classification problem.

Another example of a multi-class classification problem is predicting what language a website is in from the text on the website. The classes here would be a pre-defined list of possible languages.

For regression tasks, the goal is to predict a continuous number, or a *floating point number* in programming terms (a real number in mathematical terms). Predicting a person's annual income from their education, their age and where they live, is a[n example of a] regression task. When predicting income, the predicted value is an *amount*, and can be any number in a given range. Another example of a regression task is predicting the yield of a corn farm, given attributes such as previous yields, weather and number of employees working on the farm. The yield again can be an arbitrary number.

An easy way to distinguish between classification and regression tasks is to ask whether there is some kind of ordering or continuity in the output. If there is an ordering, or a continuity between possible outcomes, then the problem is a regression problem.

Think about predicting annual income. There is a clear ordering of "making more money" or "making less money". There is a natural understanding that 40.000$ per year is *between* 50.000$ per year and 30.000$ per year. There is also a continuity in the output. Whether a person makes 40,000$ or 40,001$ a year does not make a tangible difference, even though they are different amounts of money. So if our algorithm predicts 39,999$ or 40,001$ when it should have predicted 40,000$, we don't mind that much.

Contrastively, for the task of recognizing the language of a website (which is a classification problem), there is no matter of degree. A website is in one language, or it is in another. There is no continuity between languages, and there is no language that is *between* English and French [footnote: We ask linguists to excuse the simplified presentation of languages as distinct and fixed entities].

Generalization, Overfitting and Underfitting

In supervised learning, we want to built a model on the training data, and then be able to make accurate predictions on new, unseen data, that has the same characteristics as the training set that we used. If a model is able to make accurate predictions on unseen data, we say it is able to *generalize* from the training set to the test set.

We want to build a model that is able to generalize as well as possible.

Usually we build a model in such a way that it can make accurate predictions on the training set. If the training and test set have enough in common, we expect the model to also be accurate on the test set.

However, there are some cases where this can go wrong. For example, if we allow ourselves to build very complex models, we can always be as accurate as we like on the training set.

Let's take a look at a made-up example. Say a novice data scientist wants to predict a person's salary, and for each person, the only characteristic he has is the date of birth. The dataset might look like this:

|Date of Birth|Annual salary ($)|

|-|-|

|30/4/1950|50500|

|05/8/1964|41000|

|09/2/2001|35200|

|17/5/1989|36000|

Because our novice data scientist knows he needs to present a machine learning algorithm with numbers, he replaces the date of birth with each persons age at the time of analysis, in 2016. That seems very little to go by, so our novice data scientist also adds the last four digits of their social security number, their house number, their zip code, and the number of their children.

Now the data looks like this:

|Age|SSN|House|ZIP|Children|Annual salary ($)|

|-|-|-|-|-|-|

|66|1882|19|10030|2|50500|

|52|1337|2|10028|0|41000|

|22|3467|8|10041|1|35200|

|25|8391|27|10009|4|36000|

Now he builds a machine learning model using the first three rows as a training set. Let's save how the algorithm works for later. The algorithm produces the following formula for the annual salary:

```
salary = 333 * x[0] + 1 * x[1] + 237 * x[2] - 20 * x[3] + 26 * x[4] +
225866
```

Here `x[0]` to `x[4]` contain the age, last digits of the SSN, the house number, ZIP code and number of children.

The formula works very well on the training set, the first three rows of the dataset. The predictions for the training set are 53681, 44433 and 37761 which are very close to the true values.

However, the prediction the formula makes for the fourth point in the dataset, which was not part of the training set, is 48905, which is quite far from the 36000 which was the desired output.

So what happened here? The data scientist allowed his machine learning algorithm to build a relatively complex interaction between the five features and the output (the annual salary) without a lot of support for this model in the data. The result is a model that doesn't reflect a real world relationship. For example, this model predicts that you would make $237 more if you move to the house next door (237 is the coefficient for `x[2]`!

Building a complex model that does well on the training set but does not generalize to new data is known as *overfitting*, because we are focusing too much on the particularities of the training data. Avoiding overfitting is a crucial aspect of building a successful machine learning model. A good way to avoid overfitting is to restrict ourselves to building very simple models.

A much simpler model for the salary prediction task is to always predict the average salary of the three people in the training set, which is

Predicting that everybody's salary is 42233 is clearly too simple, and does not capture the variation in our training set very well. Using too simple a model is called *underfitting*, because we don't explain the target output for the training data well enough.

A middle ground for the salary prediction would be to use age as a single feature, which restricts us to very simple models, but still allows us to capture some trends in our data.

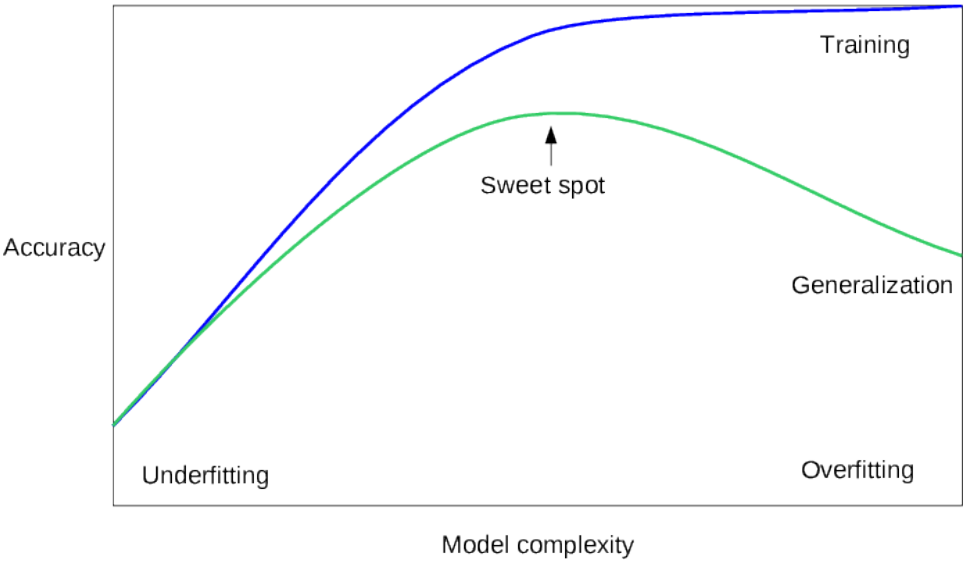
A model only including the age feature is:

```
salary = 323 * age + 27146
```

This model makes predictions of 48464, 43942, and 34252 for the training set, which is not as good as our previous model.

However, it generalizes much better to the test set when compared to the complex model we used before. It predicts 35221 for the fourth row in the table.

The trade-off between overfitting and underfitting is illustrated in Figure model_complexity.



If we choose use a model that is too simple, we will do badly on the training set, and similarly badly on the test set, as we would using only the mean prediction.

The more complex we allow our model to be, the better we will be able to predict on the training data. However, if our model becomes too complex, we start focusing too much on the particularities of our training set, and the model will not generalize well to new data.

There is a sweet spot in between, which will yield the best generalization performance. This is the model we want to find.

Understanding the implications of model complexity is hard, and has different implications for each kind of machine learning model.

Supervised Machine Learning Algorithms

We will now go through the most popular machine learning algorithms and explain how they learn from data and how they make predictions. We will also discuss how the concept of model complexity plays out for each of these models.

While an in-depth discussion of each algorithm is beyond the scope of this book, we will try to give some intuition about how each algorithm builds a model.

We will also discuss strength and weaknesses of each algorithm, and what kind of data they can be best applied to. We will also explain the meaning of the most important parameters and options. Discussing all of them is beyond the scope of the book, and we refer you to the scikit-learn documentation for more details.

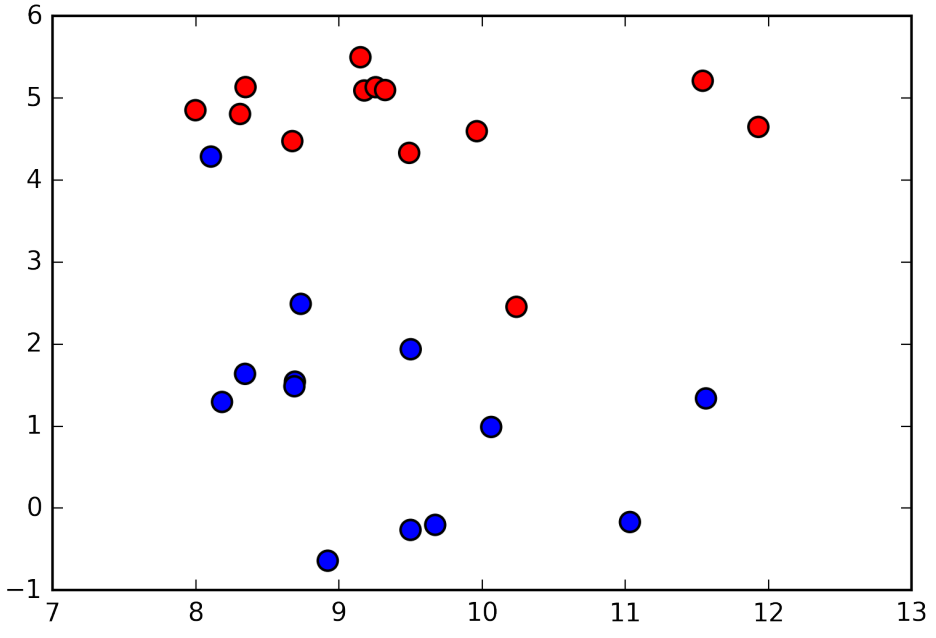
Many algorithms have a classification and a regression variant, and we will describe both.

It is not necessary to read through the description of each algorithm in detail, but understanding the models will give you a better feeling for the different ways machine learning algorithms can work. This chapter can also be used as a reference guide, and you can come back to it when you are unsure about the workings of any of the algorithms.

We will use several datasets to illustrate the different algorithms. Some of the datasets will be small synthetic (meaning made-up) datasets, designed to highlight particular aspects of the algorithms. Other datasets will be larger, real world examples datasets.

An example of a synthetic two-class classification dataset is the `forge` dataset, which has two features. Below is a scatter plot visualizing all of the data points in this dataset. The plot has the first feature on the x-axis and the second feature on the y-axis. As is always the case in in scatter plots, each data point is represented as one dot. The color of the dot indicates its class, with red meaning class 0 and blue meaning class 1.

```
X, y = mglearn.datasets.make_forge()
plt.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm2)
print("X.shape: %s" % (X.shape,))
```



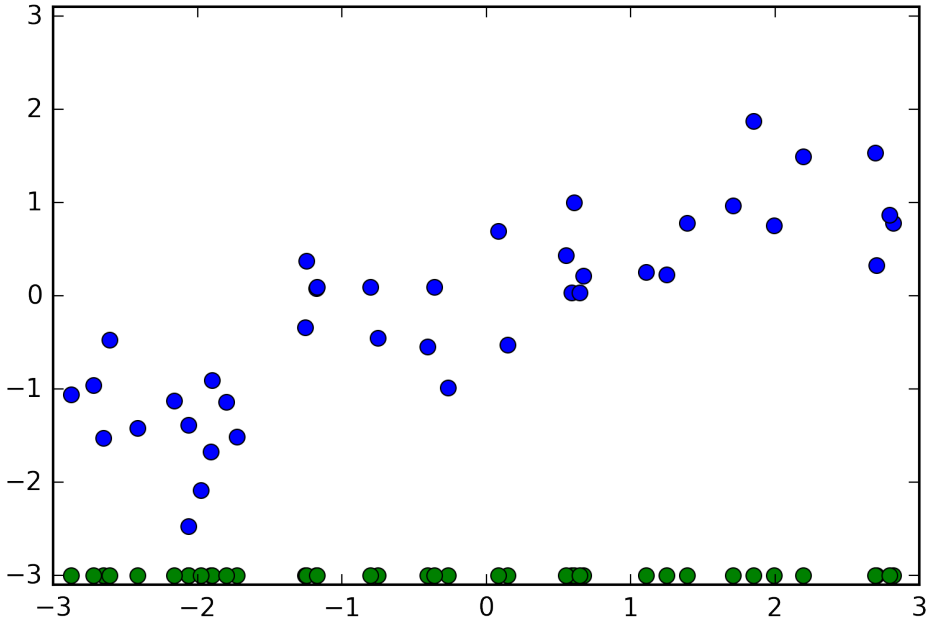

```
X.shape: (26, 2)
```

As you can see from `X.shape`, this dataset consists of 26 data points, with 2 features.

To illustrate regression algorithms, we will use the synthetic `wave` dataset shown below. The `wave dataset` only has a single input feature, and a continuous target variable (or *response*) that we want to model.

The plot below is showing the single feature on the x-axis, with the data points as green dots. For each data point, the target output is plotted in blue on the y-axis.

```
X, y = mglearn.datasets.make_wave(n_samples=40)

plt.plot(X, y, 'o')
plt.plot(X, -3 * np.ones(len(X)), 'o')
plt.ylim(-3.1, 3.1)
```



We are using these very simple, low-dimensional datasets as we can easily visualize them -- a computer monitor has two dimensions, so data with more than two features is hard to show. Any intuition derived from datasets with few features (also called *low-dimensional* datasets) might not hold in datasets with many features (*high dimensional* datasets). As long as you keep that in mind, inspecting algorithms on low-dimensional datasets can be very instructive.

We will complement these small synthetic dataset with two real-world datasets that are included in scikit-learn. One is the Wisconsin breast cancer dataset (or `cancer` for short), which records clinical measurements of breast cancer tumors. Each tumor is labeled as "benign" (for harmless tumors) or "malignant" (for cancerous tumors), and the task is to learn to predict whether a tumor is malignant based on the measurements of the tissue.

The data can be loaded using the `load_breast_cancer` from scikit-learn. Datasets that are included in scikit-learn are usually stored as `Bunch` objects, which contain some information about the dataset as well as the actual data.

All you need to know about `Bunch` objects is that they behave like dictionaries, with the added benefit that you can access values using a dot (as in `bunch.key` instead of `bunch['key']`).

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
cancer.keys()
```

```
dict_keys(['DESCR', 'target_names', 'data', 'target', 'feature_names'])
```

The dataset consists of 569 data points, with 30 features each:

```
print(cancer.data.shape)
```

```
(569, 30)
```

Of these 569 data points, 212 are labeled as malignant, and 357 as benign:

```
print(cancer.target_names)
np.bincount(cancer.target)
```

```
array([212, 357])
```

```
['malignant' 'benign']
```

To get a description of the semantic meaning of each feature, we can have a look at the `feature_names` attribute:

```
cancer.feature_names
```

```
array(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal dimension',
       'radius error', 'texture error', 'perimeter error', 'area error',
       'smoothness error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error', 'fractal dimension error',
       'worst radius', 'worst texture', 'worst perimeter', 'worst area',
       'worst smoothness', 'worst compactness', 'worst concavity',
       'worst concave points', 'worst symmetry', 'worst fractal dimension'],
      dtype='<U23')
```

You can find out more about the data by reading `cancer.DESCR` if you are interested.

We will also be using a real-world regression dataset, the Boston Housing dataset. The task associated with this dataset is to predict the median value of homes in several Boston neighborhoods in the 1970s, using information about the neighborhoods such as crime rate, proximity to the Charles River, highway accessibility and so on.

The datasets contains 506 data points, described by 13 features:

```
from sklearn.datasets import load_boston
boston = load_boston()
print(boston.data.shape)
```

```
(506, 13)
```

Again, you can get more information about the dataset by reading the `DESCR` attribute of `boston`.

For our purposes here, we will actually expand this dataset, by not only considering these 13 measurements as input features, but also looking at all products (also called *interactions*) between features.

In other words, we will not only consider crime rate and highway accessibility as a feature, but also the product of crime rate and highway accessibility. Including derived feature like these is called *feature engineering*, which we will discuss in more detail in Chapter 5 (Representing Data).

This derived dataset can be loaded using the `load_extended_boston` function:

```
X, y = mglearn.datasets.load_extended_boston()
print(X.shape)
```

```
(506, 105)
```

The resulting 105 features are the 13 original features, the 13 choose 2 = 91 (Footnote: the number of ways to pick 2 elements out of 13 elements) features that are product of two features, and one constant feature.

We will use these datasets to explain and illustrate the properties of the different machine learning algorithms. But for now, let's get to the algorithms themselves. First, we will revisit the k-Nearest Neighbor algorithm, that we already saw in the last chapter.

k-Nearest Neighbor

The k-Nearest Neighbors (kNN) algorithm is arguably the simplest machine learning algorithm. Building the model only consists of storing the training dataset. To make a prediction for a new data point, the algorithm finds the closest data points in the training dataset, it "nearest neighbors".

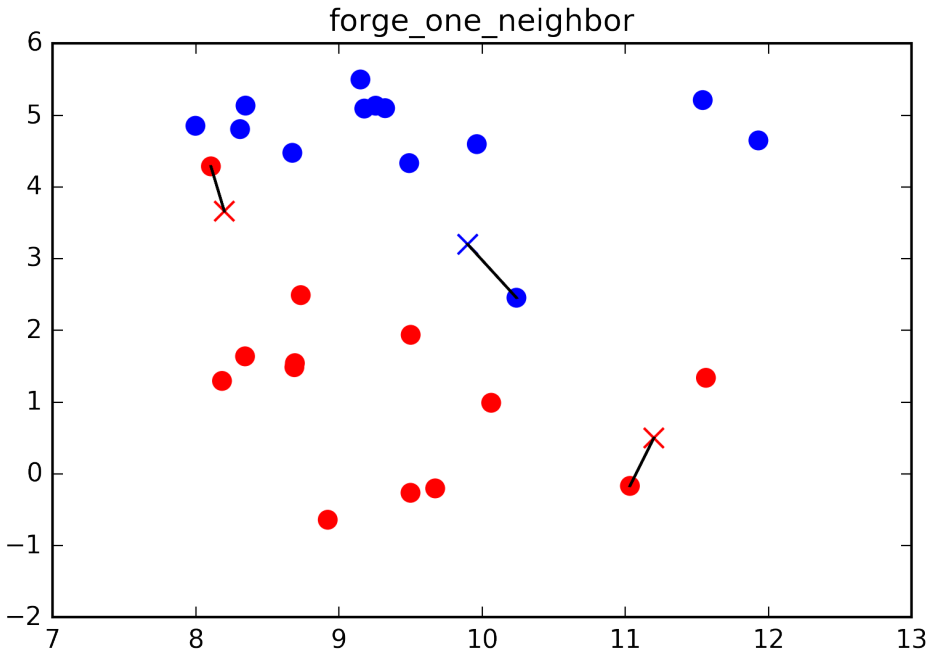
k-Neighbors Classification

In its simplest version, the algorithm only considers exactly one nearest neighbor, which is the closest training data point to the point we want to make a prediction for.

The prediction is then simply the known output for this training point.

Figure forge_one_neighbor illustrates this for the case of classification on the `forge` dataset.

```
mglearn.plots.plot_knn_classification(n_neighbors=1)
plt.title("forge_one_neighbor");
```

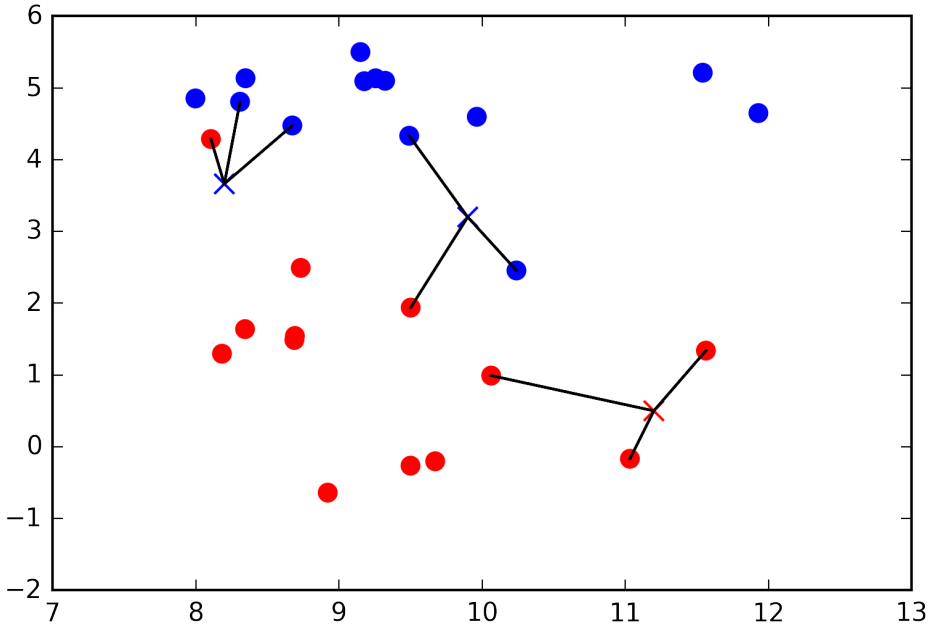


Here, we added three new data points, shown as crosses. For each of them, we marked the closest point in the training set. The prediction of the one-nearest-neighbor algorithm is the label of that point (shown by the color of the cross).

Instead of considering only the closest neighbor, we can also consider an arbitrary number k of neighbors. This is where the name of the k neighbors algorithm comes from. When considering more than one neighbor, we use *voting* to assign a label. This means, for each test point, we count how many neighbors are red, and how many neighbors are blue. We then assign the class that is more frequent: in other words, the majority class among the k neighbors.

Below is an illustration using the three closest neighbors. Again, the prediction is shown as the color of the cross. You can see that the prediction changed for the point in the top left from using only one neighbor.

```
mglearn.plots.plot_knn_classification(n_neighbors=3)
```



While this illustration is for a binary classification problem, you can imagine this working with any number of classes. For more classes, we count how many neighbors belong to each class, and again predict the most common class.

Now let's look at how we can apply the k nearest neighbors algorithm using scikit-learn.

First, we split our data into a training and a test set, so we can evaluate generalization performance, as discussed in Chapter 1 (Introduction).

```
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Next we import and instantiate the class. This is when we can set parameters, like the number of neighbors to use. Here, we set it to three.

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

Now, we fit the classifier using the training set. For `KNeighborsClassifier` this means storing the dataset, so we can compute neighbors during prediction.

```
clf.fit(X_train, y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
```

```
                metric_params=None, n_jobs=1, n_neighbors=3, p=2,

                weights='uniform')
```

To make predictions on the test data, we call the `predict` method. This computes the nearest neighbors in the training set and finds the most common class among these:

```
clf.predict(X_test)
```

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

To evaluate how well our model generalizes, we can call the `score` method with the test data together with the test labels:

```
clf.score(X_test, y_test)
```

```
0.8571428571428571
```

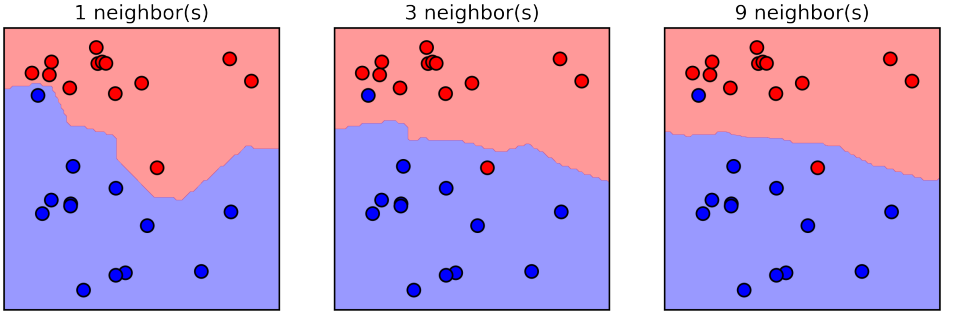
We see that our model is about 86% accurate, meaning the model predicted the class correctly for 85% of the samples in the test dataset.

Analyzing KNeighborsClassifier

For two-dimensional datasets, we can also illustrate the prediction for all possible test point in the xy-plane. We color the plane red in regions where points would be assigned the red class, and blue otherwise. This lets us view the *decision boundary*, which is the divide between where the algorithm assigns class red versus where it assigns class blue.

Here is a visualization of the decision boundary for one, three and five neighbors:

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))

for n_neighbors, ax in zip([1, 3, 9], axes):
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
    ax.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm2)
    ax.set_title("%d neighbor(s)" % n_neighbors)
```



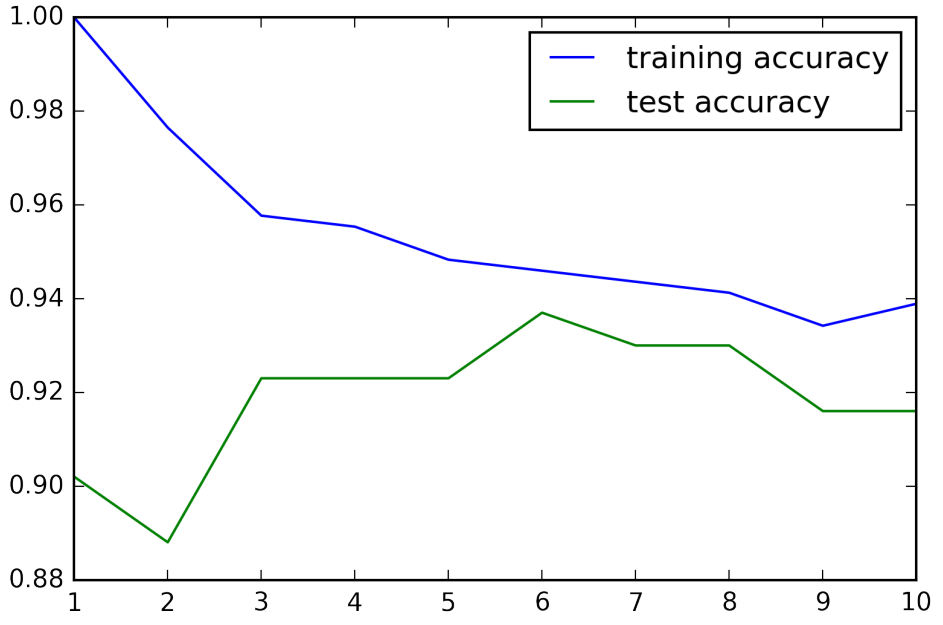
As you can see in the left figure, using a single neighbor results in a decision boundary that follows the training data closely. Considering more and more neighbors leads to a smoother decision boundary. A smoother boundary corresponds to a simple model. In other words, using few neighbors corresponds to high model complexity (as shown on the right side of Figure model_complexity), and using many neighbors corresponds to low model complexity (as shown on the left side of Figure model_complexity).

Let's investigate whether we can confirm the connection between model complexity and generalization that we discussed above.

We will do this on the real world breast cancer dataset.

We begin by splitting the dataset into a training and a test set. Then we will evaluate training and test set performance with different numbers of neighbors.

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=66)

training_accuracy = []
test_accuracy = []
# try n_neighbors from 1 to 10.
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    # build the model
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(clf.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
plt.legend()
```



The plot shows the training and test set accuracy on the y axis against the setting of `n_neighbors` on the x axis. While the real world plots are rarely very smooth, we can still recognize some of the characteristics of overfitting and underfitting. As considering fewer neighbors corresponds to a more complex model, the plot is horizontally flipped relative to the illustration in Figure model_complexity.

Considering a single nearest neighbor, the prediction on the training set is perfect. Considering more neighbors, the model becomes more simple, and the training accuracy drops.

The test set accuracy for using a single neighbor is lower then when using more neighbors, indicating that using a single nearest neighbor leads to a model that is too complex. On the other hand, when considering 10 neighbors, the model is too simple, and performance is even worse. The best performance is somewhere in the middle, around using six neighbors.

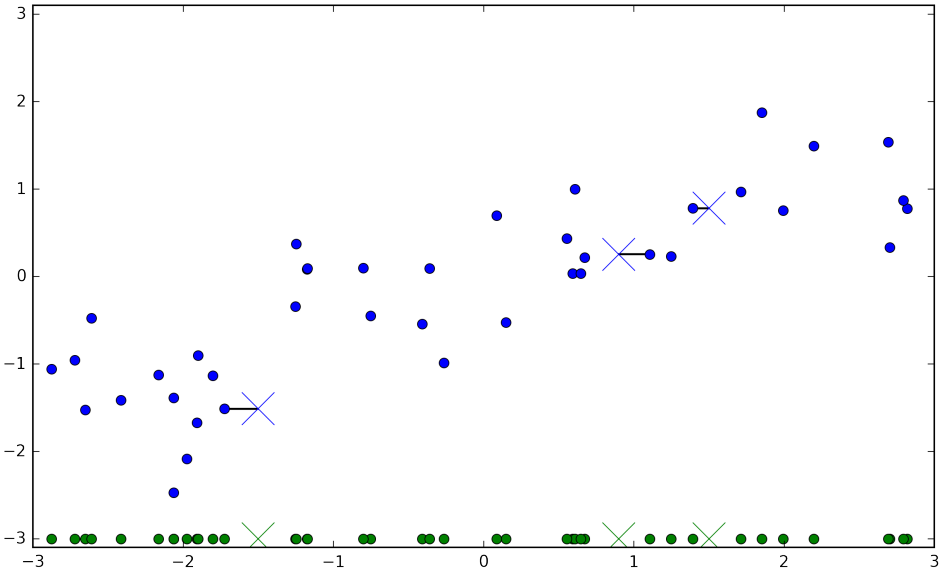
Still, it is good to keep the scale of the plot in mind. The worst performance is around 88% accuracy, which might still be acceptable.

k-Neighbors Regression

There is also a regression variant of the k-nearest neighbors algorithm. Again, let's start by using a single nearest neighbor, this time using the `wave` dataset. We added three test data points as green crosses on the x axis.

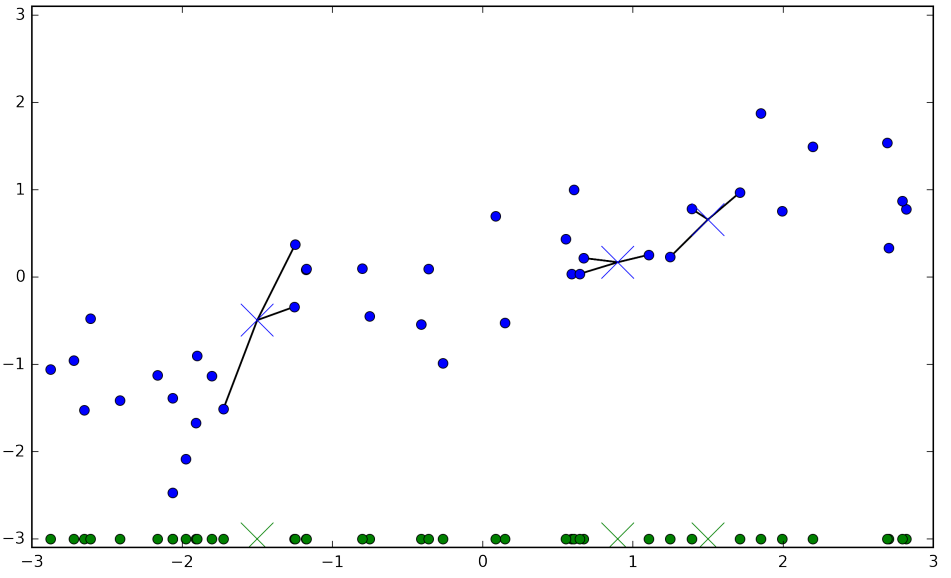
The prediction using a single neighbor is just the target value of the nearest neighbor, shown as the blue cross:

```
mglearn.plots.plot_knn_regression(n_neighbors=1)
```



Again, we can also use more than one nearest neighbor for regression. When using multiple nearest neighbors for regression, the prediction is the average (or mean) of the relevant neighbors:

```
mglearn.plots.plot_knn_regression(n_neighbors=3)
```



The k nearest neighbors algorithm for regression is implemented in the `KNeighborsRegressor` class in scikit-learn.

Using it looks much like the `KNeighborsClassifier` above:

```
from sklearn.neighbors import KNeighborsRegressor

X, y = mglearn.datasets.make_wave(n_samples=40)

# split the wave dataset into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# Instantiate the model, set the number of neighbors to consider to 3:
reg = KNeighborsRegressor(n_neighbors=3)
# Fit the model using the training data and training targets:
reg.fit(X_train, y_train)
```

```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',

          metric_params=None, n_jobs=1, n_neighbors=3, p=2,

          weights='uniform')
```

Now, we can make predictions on the test set:

```
reg.predict(X_test)
```

```
array([-0.05396539,  0.35686046,  1.13671923, -1.89415682, -1.13881398,

       -1.63113382,  0.35686046,  0.91241374, -0.44680446, -1.13881398])
```

We can also evaluate the model using the score method, which for regressors returns the R^2 score.

The R^2 score, also known as coefficient of determination, is a measure of goodness of a prediction for a regression model, and yields a score up to 1. A value of 1 corresponds to a perfect prediction, and a value of 0 corresponds to a constant model that just predicts the mean of the training set responses `y_train`.

```
reg.score(X_test, y_test)
```

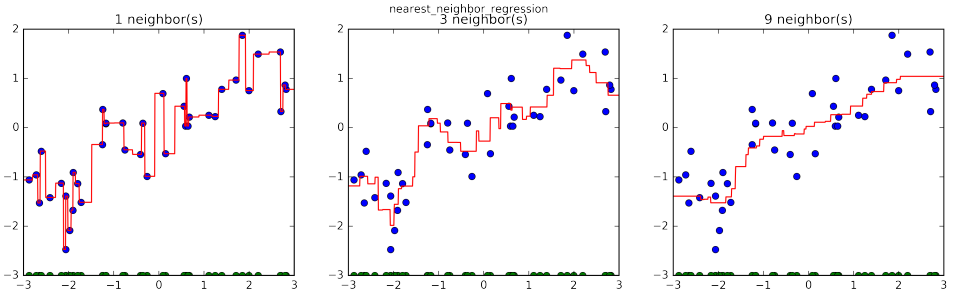
```
0.83441724462496036
```

Here, the score is `0.83` which indicates a relatively good model fit.

Analyzing k nearest neighbors regression

For our one-dimensional dataset, we can see what the predictions look like for all possible feature values. To do this, we create a test-dataset consisting of many points on the line.

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
# create 1000 data points, evenly spaced between -3 and 3
line = np.linspace(-3, 3, 1000).reshape(-1, 1)
plt.suptitle("nearest_neighbor_regression")
for n_neighbors, ax in zip([1, 3, 9], axes):
    # make predictions using 1, 3 or 9 neighbors
    reg = KNeighborsRegressor(n_neighbors=n_neighbors).fit(X, y)
    ax.plot(X, y, 'o')
    ax.plot(X, -3 * np.ones(len(X)), 'o')
    ax.plot(line, reg.predict(line))
    ax.set_title("%d neighbor(s)" % n_neighbors)
```



In the plots above, the blue points are again the responses for the training data, while the red line is the prediction made by the model for all points on the line.

Using only a single neighbor, each point in the training set has an obvious influence on the predictions, and the predicted values go through all of the data points. This leads to a very unsteady prediction. Considering more neighbors leads to smoother predictions, but these do not fit the training data as well.

Strengths, weaknesses and parameters

In principal, there are two important parameters to the KNeighbors classifier: the number of neighbors and how you measure distance between data points. In practice, using a small number of neighbors like 3 or 5 often works well, but you should certainly adjust this parameter. Choosing the right distance measure is somewhat beyond the scope of this book. By default, Euclidean distance is used, which works well in many settings.

One of the strengths of nearest neighbors is that the model is very easy to understand, and often gives reasonable performance without a lot of adjustments. Using nearest neighbors is a good baseline method to try before considering more advanced techniques. Building the nearest neighbors model is usually very fast, but when your training set is very large (either in number of features or in number of samples) prediction can be slow.

When using nearest neighbors, it's important to preprocess your data (see Chapter 3 Unsupervised Learning). Nearest neighbors often does not perform well on dataset with very many features, in particular sparse datasets, a common type of data in which there are many features, but only few of the features are non-zero for any given data point.

So while the nearest neighbors algorithm is easy to understand, it is not often used in practice, due to prediction being slow, and its inability to handle many features. The method we discuss next has neither of these drawbacks.

Linear models

Linear models are a class of models that are widely used in practice, and have been studied extensively in the last few decades, with roots going back over a hundred years.

Linear models are models that make a prediction that using a *linear function* of the input features, which we will explain below.

Linear models for regression

For regression, the general prediction formula for a linear model looks as follows:

\begin{align*}

&\hat{y} = w[0] *x[0]* + *w[1]* x[1] + \dotsc + w[p] *x[p]* + *b* \text{ (1) linear regression}

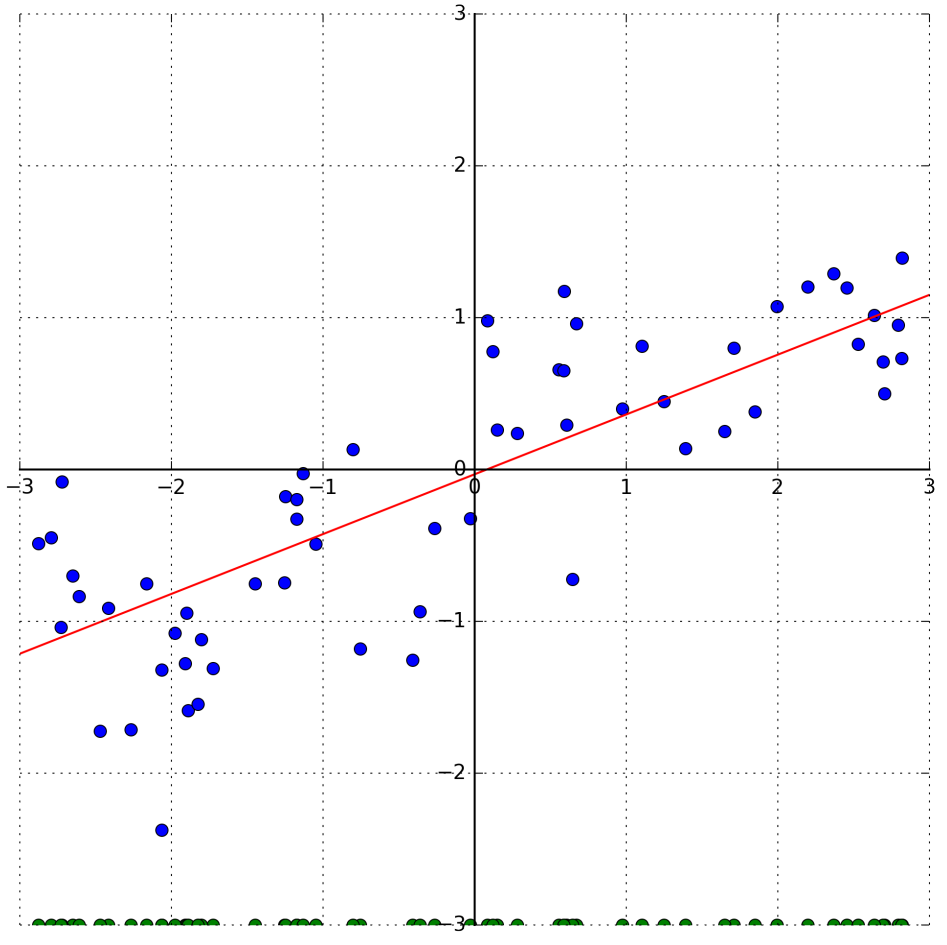
\end{align*}

Here, $x[0]$ to $x[p]$ denotes the features (here the number of features is p) of a single data point, w and b are parameters of the model that are learned, and $\hat{y}$ is the prediction the model makes. For a dataset with a single feature, this is

which you might remember as the equation for a line from high school mathematics. Here, $w[0]$ is the slope, and b is the y-axis offset. For more features, `w` contains the slopes along each feature axis. Alternatively, you can think of the predicted response as being a weighted sum of the input features, with weights (which can be negative) given by the entries of `w`.

Trying to learn the parameters $w[0]$ and b on our one-dimensional `wave` dataset might lead to the following line:

```
mglearn.plots.plot_linear_regression_wave()
```



```
w[0]: 0.393906  b: -0.031804
```

We added a coordinate cross into the plot to make it easier to understand the line. Looking at `w[0]` we see that the slope should be roughly around .4, which we can confirm visually in the plot above. The intercept is where the prediction line should cross the y-axis, which is slightly below 0, which you can also confirm in the image.

Linear models for regression can be characterized as regression models for which the prediction is a line for a single feature, a plane when using two features, or a hyperplane in higher dimensions (that is when having more features).

If you compare the predictions made by the red line with those made by the KNeighborsRegressor in Figure nearest_neighbor_regression, using a straight line to make predictions seems very restrictive. It looks like all the fine details of the data are lost. In a sense this is true. It is a strong (and somewhat unrealistic) assumption that our target y is a linear combination of the features. But looking at one-dimensional data gives a somewhat skewed perspective. For datasets with many features, linear models can be very powerful. In particular, if you have more features than training data points, any target y can be perfectly modeled (on the training set) as a linear function (FOOTNOTE This is easy to see if you know some linear algebra).

There are many different linear models for regression. The difference between these models is how the model parameters w and b are learned from the training data, and how model complexity can be controlled. We will now go through the most popular linear models for regression.

Linear Regression aka Ordinary Least Squares

Linear regression or *Ordinary Least Squares* (OLS) is the simplest and most classic linear method for regression.

Linear regression finds the parameters w and b that minimize the *mean squared error* between predictions and the true regression targets y on the training set. The mean squared error is the sum of the squared differences between the predictions and the true values. Linear regression has no parameters, which is a benefit, but it also has no way to control model complexity.

Here is the code that produces the model you can see in figure XX.

```
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

lr = LinearRegression().fit(X_train, y_train)
```

The "slope" parameters `w`, also called weights or *coefficients* are stored in the `coef_` attribute, while the offset or *intercept* `b` is stored in the `intercept_` attribute. [Footnote: you might notice the strange-looking trailing underscore. Scikit-learn always

stores anything that is derived from the training data in attributes that end with a trailing underscore. That is to separate them from parameters that are set by the user.]

```
print("lr.coef_: %s" % lr.coef_)
print("lr.intercept_: %s" % lr.intercept_)
```

```
lr.coef_: [ 0.39390555]

lr.intercept_: -0.0318043430268
```

The `intercept_` attribute is always a single float number, while the `coef_` attribute is a numpy array with one entry per input feature. As we only have a single input feature in the `wave` dataset, `lr.coef_` only has a single entry.

Let's look at the training set and test set performance:

```
print("training set score: %f" % lr.score(X_train, y_train))
print("test set score: %f" % lr.score(X_test, y_test))
```

```
training set score: 0.670089

test set score: 0.659337
```

An R^2 of around .66 is not very good, but we can see that the score on training and test set are very close together. This means we are likely underfitting, not overfitting. For this one-dimensional dataset, there is little danger of overfitting, as the model is very simple (or restricted).

However, with higher dimensional datasets (meaning a large number of features), linear models become more powerful, and there is a higher chance of overfitting.

Let's take a look at how `LinearRegression` performs on a more complex dataset, like the Boston Housing dataset. Remember that this dataset has 506 samples and 105 derived features.

We load the dataset and split it into a training and a test set. Then we build the linear regression model as before:

```
X, y = mglearn.datasets.load_extended_boston()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

When comparing training set and test set score, we find that we predict very accurately on the training set, but the R^2 on the test set is much worse:

```
print("training set score: %f" % lr.score(X_train, y_train))
print("test set score: %f" % lr.score(X_test, y_test))
```

```
training set score: 0.952353

test set score: 0.605775
```

This is a clear sign of overfitting, and therefore we should try to find a model that allows us to control complexity.

One of the most commonly used alternatives to standard linear regression is *Ridge regression*, which we will look into next.

Ridge regression

Ridge regression is also a linear model for regression, so the formula it uses to make predictions is still Formula (1), as for ordinary least squares. In Ridge regression,the coefficients `w` are chosen not only so that they predict well on the training data, but there is an additional constraint. We also want the magnitude of coefficients to be as small as possible; in other words, all entries of `w` should be close to `0`.

Intuitively, this means each feature should have as little effect on the outcome as possible (which translates to having a small slope), while still predicting well.

This constraint is an example of what is called *regularization*. Regularization means explicitly restricting a model to avoid overfitting. The particular kind used by Ridge regression is known as `l2` regularization. [footnote: Mathematically, Ridge penalizes the l2 norm of the coefficients, or the Euclidean length of w.]

Ridge regression is implemented in `linear_model.Ridge`. Let's see how well it does on the extended Boston dataset:

```
from sklearn.linear_model import Ridge

ridge = Ridge().fit(X_train, y_train)
print("training set score: %f" % ridge.score(X_train, y_train))
print("test set score: %f" % ridge.score(X_test, y_test))
```

```
training set score: 0.886058

test set score: 0.752714
```

As you can see, the training set score of Ridge is *lower* than for LinearRegression, while the test set score is *higher*. This is consistent with our expectation. With linear regression, we were overfitting to our data. Ridge is a more restricted model, so we are less likely to overfit. A less complex model means worse performance on the training set, but better generalization.

As we are only interested in generalization performance, we should choose the `Ridge` model over the `LinearRegression` model.

The Ridge model makes a trade-off between the simplicity of the model (near zero coefficients) and its performance on the training set. How much importance the model places on simplicity versus training set performance can be specified by the user, using the `alpha` parameter. Above, we used the default parameter `alpha=1.0`. There is no reason why this would give us the best trade-off, though. Increasing alpha

forces coefficients to move more towards zero, which decreases training set performance, but might help generalization.

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("training set score: %f" % ridge10.score(X_train, y_train))
print("test set score: %f" % ridge10.score(X_test, y_test))
```

```
training set score: 0.788346

test set score: 0.635897
```

Decreasing alpha allows the coefficients to be less restricted, meaning we move right on the figure XXX.

For very small values of alpha, coefficients are barely restricted at all, and we end up with a model that resembles `LinearRegression`.

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("training set score: %f" % ridge01.score(X_train, y_train))
print("test set score: %f" % ridge01.score(X_test, y_test))
```

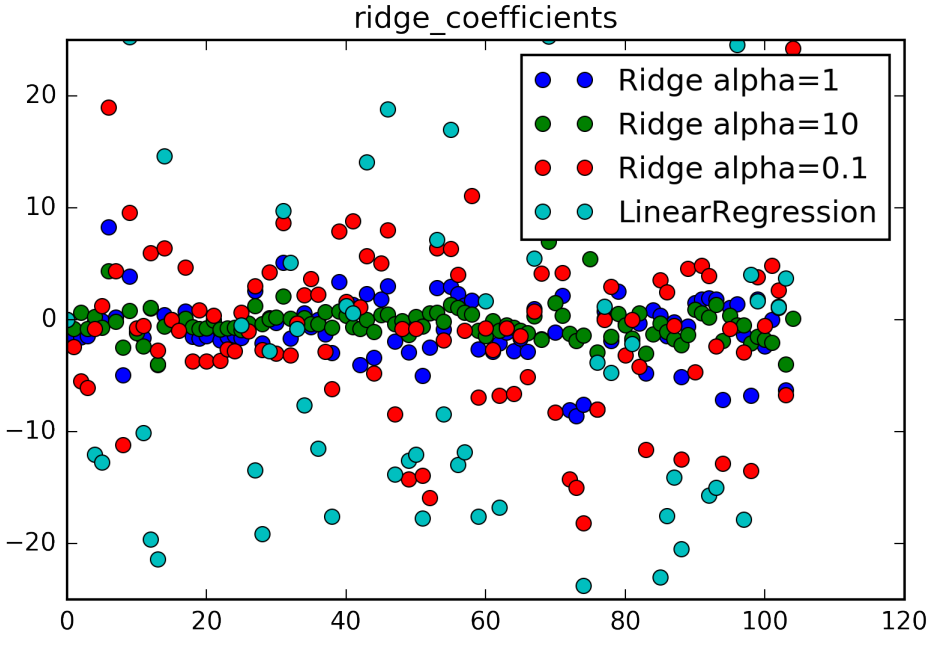
```
training set score: 0.928578

test set score: 0.771793
```

Here, `alpha=0.1` seems to be working well. We could try decreasing `alpha` even more to improve generalization. For now, notice how the parameter `alpha` corresponds to the model complexity as shown in Figure model_complexity. We will discuss methods to properly select parameters in Chapter 6 (Model Selection).

We can also get a more qualitative insight into how the `alpha` parameter changes the model by inspecting the `coef_` attribute of models with different values of `alpha`. A higher alpha means a more restricted model, so we expect that the entries of `coef_` have smaller magnitude for a high value of alpha than for a low value of alpha.

This is confirmed in the plot below:

```
plt.title("ridge_coefficients")
plt.plot(ridge.coef_, 'o', label="Ridge alpha=1")
plt.plot(ridge10.coef_, 'o', label="Ridge alpha=10")
plt.plot(ridge01.coef_, 'o', label="Ridge alpha=0.1")

plt.plot(lr.coef_, 'o', label="LinearRegression")
plt.ylim(-25, 25)
plt.legend()
```



Here, the x-axis enumerates the entries of `coef_`: `x=0` shows the coefficient associated with the first feature, `x=1` the coefficient associated with the second feature, and so on up to x=100. The y-axis shows the numeric value of the corresponding value of the coefficient. The main take-away here is that for alpha=10 (as shown by the green dots), the coefficients are mostly between around -3 and 3. The coefficients for the ridge model with alpha=1 (as shown by the blue dots), are somewhat larger. The red dots have larger magnitude still, and many of the teal dots, corresponding to linear regression without any regularization (which would be alpha=0) are so large they are even outside of the chart.

Lasso

An alternative to Ridge for regularizing linear regression is the *Lasso*. The lasso also restricts coefficients to be close to zero, similarly to Ridge regression, but in a slightly different way, called "l1" regularization.[footnote: The Lasso penalizes the l1 norm of the coefficient vector, or in other words the sum of the absolute values of the coefficients].

The consequence of l1 regularization is that when using the Lasso, some coefficients are *exactly zero*. This means some features are entirely ignored by the model. This can be seen as a form of automatic *feature selection*. Having some coefficients be exactly zero often makes a model easier to interpret, and can reveal the most important features of your model.

Let's apply the lasso to the extended Boston housing dataset:

```
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("training set score: %f" % lasso.score(X_train, y_train))
print("test set score: %f" % lasso.score(X_test, y_test))
print("number of features used: %d" % np.sum(lasso.coef_ != 0))
```

```
training set score: 0.293238

test set score: 0.209375

number of features used: 4
```

As you can see, the Lasso does quite badly, both on the training and the test set. This indicates that we are

underfitting. We find that it only used four of the 105 features. Similarly to Ridge, the Lasso also has a regularization parameter `alpha` that controls how strongly coefficients are pushed towards zero . Above, we used the default of `alpha=1.0`. To diminish underfitting, let's try decreasing alpha:

```
lasso001 = Lasso(alpha=0.01).fit(X_train, y_train)
print("training set score: %f" % lasso001.score(X_train, y_train))
print("test set score: %f" % lasso001.score(X_test, y_test))
print("number of features used: %d" % np.sum(lasso001.coef_ != 0))
```

```
/home/andy/checkout/scikit-learn/sklearn/linear_model/coordinate_descent.py:474: ConvergenceWarnir

  ConvergenceWarning)
```

A lower alpha allowed us to fit a more complex model, which worked better on the training and the test data. The performance is slightly better than using Ridge, and we are using only 32 of the 105 features. This makes this model potentially easier to understand.

If we set alpha too low, we again remove the effect of regularization and end up with a result similar to `LinearRegression`.

```
lasso00001 = Lasso(alpha=0.0001).fit(X_train, y_train)
print("training set score: %f" % lasso00001.score(X_train, y_train))
print("test set score: %f" % lasso00001.score(X_test, y_test))
print("number of features used: %d" % np.sum(lasso00001.coef_ != 0))
```

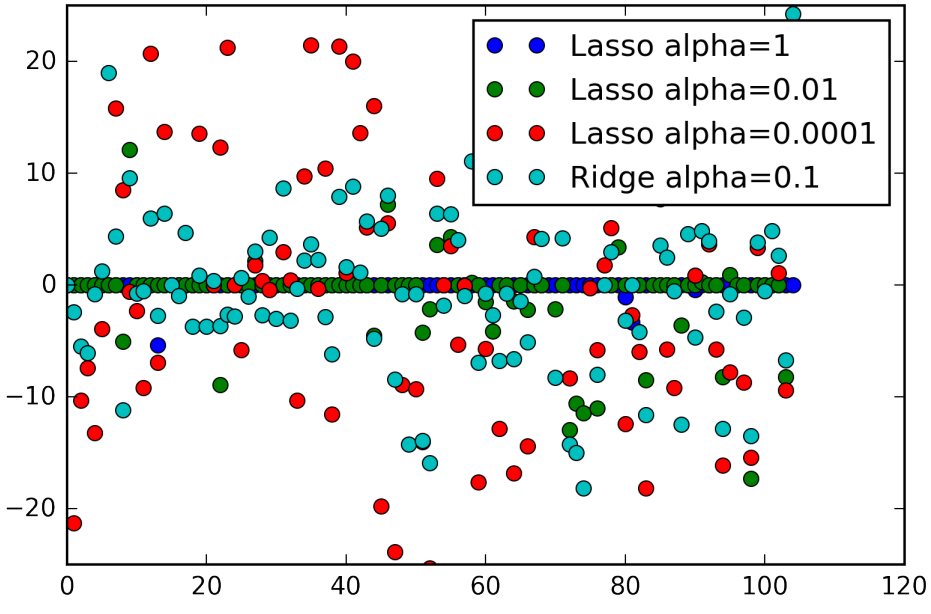
```
/home/andy/checkout/scikit-learn/sklearn/linear_model/coordinate_descent.py:474: ConvergenceWarnir

  ConvergenceWarning)
```

Again, we can plot the coefficients of the different models, similarly to Figure ridge_coefficients.

For `alpha=1`, with coefficients shown as blue dots, we not only see that most of the coefficients are zero (which we already knew), but that the remaining coefficients are also small in magnitude. Decreasing alpha to 0.01 we obtain the solution shown as the green dots, which causes most features be exactly zero. Using alpha=0.00001, we get a model that is quite unregularized, with most coefficients nonzero and of large magnitude.

For comparison, the best Ridge solution is shown in teal. The ridge model with alpha=0.1 has similar predictive performance as the lasso model with alpha=0.01, but using Ridge, all coefficients are non-zero.

```
plt.plot(lasso.coef_, 'o', label="Lasso alpha=1")
plt.plot(lasso001.coef_, 'o', label="Lasso alpha=0.01")
plt.plot(lasso00001.coef_, 'o', label="Lasso alpha=0.0001")

plt.plot(ridge01.coef_, 'o', label="Ridge alpha=0.1")
plt.ylim(-25, 25)
plt.legend()
```



In practice, Ridge regression is usually the first choice between these two models. However, if you have a large amount of features and expect only a few of them to be important, `Lasso` might be a better choice. Similarly, if you would like to have a model that is easy to interpret, `Lasso` will provide a model that is easier to understand, as it will select only a subset of the input features.

Linear models for Classification

Linear models are also extensively used for classification. Let's look at binary classification first. In this case, a prediction is made using the following formula:

\begin{align*}

&\hat{y} = w[0] *x[0]* + *w[1]* x[1] + \dotsc + w[p] * x[p] + b > 0 &\text{ (2) linear binary classification}

\end{align*}

The formula looks very similar to the one for linear regression, but instead of just returning the weighted sum of the features, we threshold the predicted value at zero. If the function was smaller than zero, we predict the class -1, if it was larger than zero, we predict the class +1.

This prediction rule is common to all linear models for classification. Again, there are many different ways to find the coefficients `w` and the intercept `b`.

For linear models for regression, the output `y` was a linear function of the features: a line, plane, or hyperplane (in higher dimensions). For linear models for classification, the *decision boundary* is a linear function of the input. In other words, a (binary) linear classifier is a classifier that separates two classes using a line, a plane or a hyperplane. We will see examples of that below.

There are many algorithms for learning linear models. These algorithms all differ in the following two ways:

1. How they measure how well a particular combination of coefficients and intercept fits the training data.

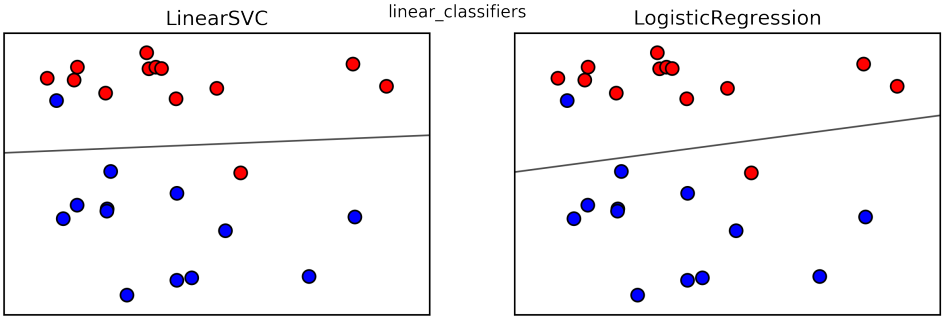
1. If and what kind of regularization they use.

Different algorithms choose different ways to measure what "fitting the training set well" means in 1. For technical mathematical reasons, it is not possible to adjust `w` and `b` to minimize the number of misclassifications the algorithms produce, as one might hope. For our purposes, and many applications, the different choices for 1. (called *loss function*) is of little significance.

The two most common linear classification algorithms are logistic regression, implemented in `linear_model.LogisticRegression` and linear support vector machines (linear SVMs), implemented in `svm.LinearSVC` (SVC stands for Support Vector Classifier). Despite its name, `LogisticRegression` is a classification algorithm and not a regression algorithm, and should not be confused with `LinearRegression`.

We can apply the `LogisticRegression` and `LinearSVC` models to the `forge` dataset, and visualize the decision boundary as found by the linear models:

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

X, y = mglearn.datasets.make_forge()

fig, axes = plt.subplots(1, 2, figsize=(10, 3))
plt.suptitle("linear_classifiers")

for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5, ax=ax, alpha=.7)
    ax.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm2)
    ax.set_title("%s" % clf.__class__.__name__)
```



In this figure, we have the first feature of the forge dataset on the x axis and the second feature on the y axis as before. We display the decision boundaries found by LinearSVC and LogisticRegression respectively as straight lines, separating the area classified as blue on the top from the area classified as red on the bottom.

In other words, any new data point that lies above the black line will be classified as blue by the respective classifier, while any point that lies below the black line will be classified as red.

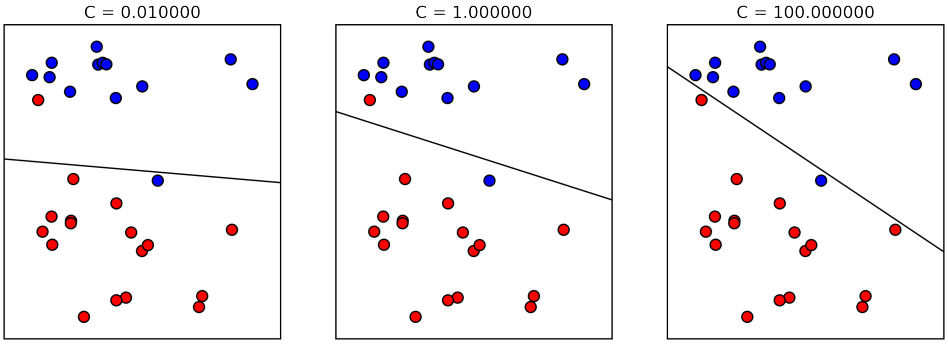
The two models come up with similar decision boundaries. Note that both misclassify two of the points. By default, both models apply an `l2` regularization, in the same way that `Ridge` does for regression.

For `LogisticRegression` and `LinearSVC` the trade-off parameter that determines the strength of the regularization is called `C`, and higher values of C correspond to *less* regularization. In other words, when using a high value of the parameter `C`, `LogisticRegression` and `LinearSVC` try to fit the training set as best as possible, while with low values of the parameter `C`, the model put more emphasis on finding a coefficient vector `w` that is close to zero.

There is another interesting intuition of how the parameter `C` acts. Using low values of `C` will cause the algorithms try to adjust to the "majority" of data points, while

using a higher value of `C` stresses the importance that each individual data point be classified correctly. Here is an illustration using `LinearSVC`.

```
mglearn.plots.plot_linear_svc_regularization()
```



On the left hand side, we have a very small `C` corresponding to a lot of regularization. Most of the blue points are at the top, and most of the red points are at the bottom. The strongly regularized model chooses a relatively horizontal line, misclassifying two points.

In the center plot, `C` is slightly higher, and the model focuses more on the two misclassified samples, tilting the decision boundary. Finally, on the right hand side, a very high value of `C` in the model tilts the decision boundary a lot, now correctly classifying all red points. One of the blue points is still misclassified, as it is not possible to correctly classify all points in this dataset using a straight line. The model illustrated on the right hand side tries hard to correctly classify all points, but might not capture the overall layout of the classes well. In other words, this model is likely overfitting.

Similarly to the case of regression, linear models for classification might seem very restrictive in low dimensional spaces, only allowing for decision boundaries which are straight lines or planes. Again, in high dimensions, linear models for classification become very powerful, and guarding against overfitting becomes increasingly important when considering more features.

Let's analyze `LinearLogistic` in more detail on the `breast_cancer` dataset:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logisticregression = LogisticRegression().fit(X_train, y_train)
print("training set score: %f" % logisticregression.score(X_train, y_train))
print("test set score: %f" % logisticregression.score(X_test, y_test))
```

```
training set score: 0.953052

test set score: 0.958042
```

The default value of `C=1` provides quite good performance, with 95% accuracy on both the training and the test set. As training and test set performance are very close, it is likely that we are underfitting. Let's try to increase `C` to fit a more flexible model.

```
logisticregression100 = LogisticRegression(C=100).fit(X_train, y_train)
print("training set score: %f" % logisticregression100.score(X_train, y_train))
print("test set score: %f" % logisticregression100.score(X_test, y_test))
```

```
training set score: 0.971831

test set score: 0.965035
```

Using `C=100` results in higher training set accuracy, and also a slightly increased test set accuracy, confirming our intuition that a more complex model should perform better.

We can also investigate what happens if we use an even more regularized model than the default of `C=1`, by setting `C=0.01`.

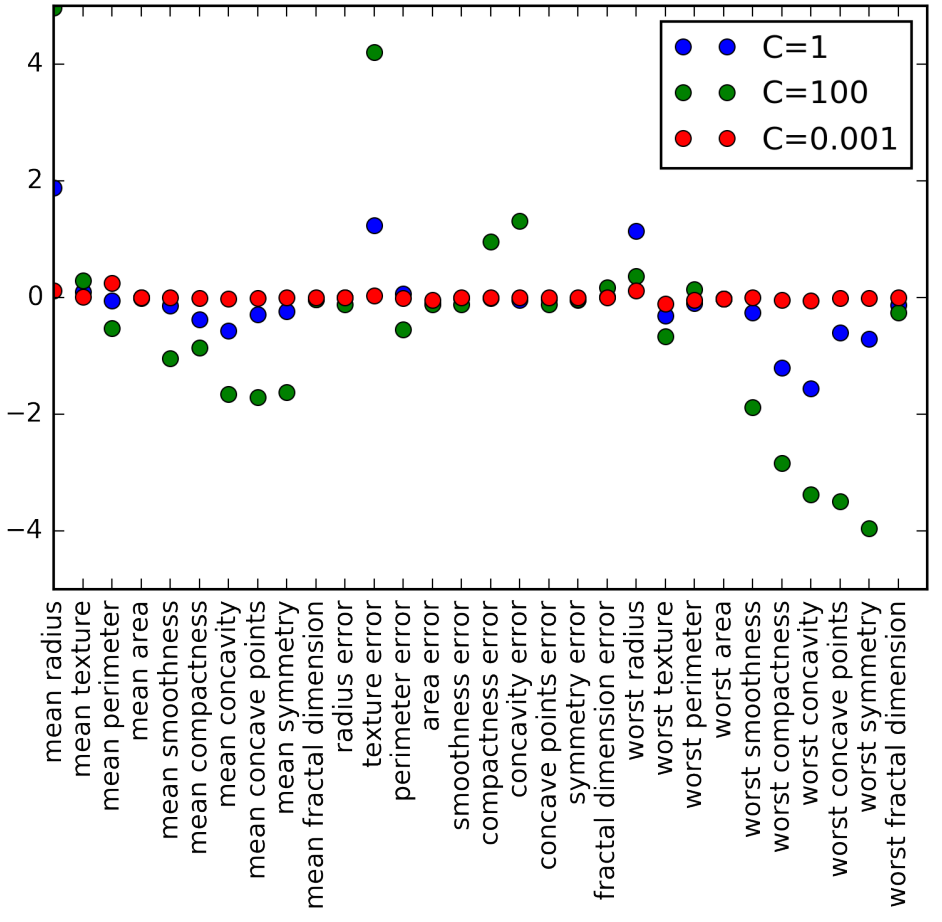
```
logisticregression001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("training set score: %f" % logisticregression001.score(X_train, y_train))
print("test set score: %f" % logisticregression001.score(X_test, y_test))
```

```
training set score: 0.934272

test set score: 0.930070
```

As expected, when moving more to the left in Figure model_complexity from an already underfit model, both training and test set accuracy decrease relative to the default parameters.

Finally, lets look at the coefficients learned by the models with the three different settings of the regularization parameter `C`.

```
plt.plot(logisticregression.coef_.T, 'o', label="C=1")
plt.plot(logisticregression100.coef_.T, 'o', label="C=100")
plt.plot(logisticregression001.coef_.T, 'o', label="C=0.001")
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.ylim(-5, 5)
plt.legend()
```

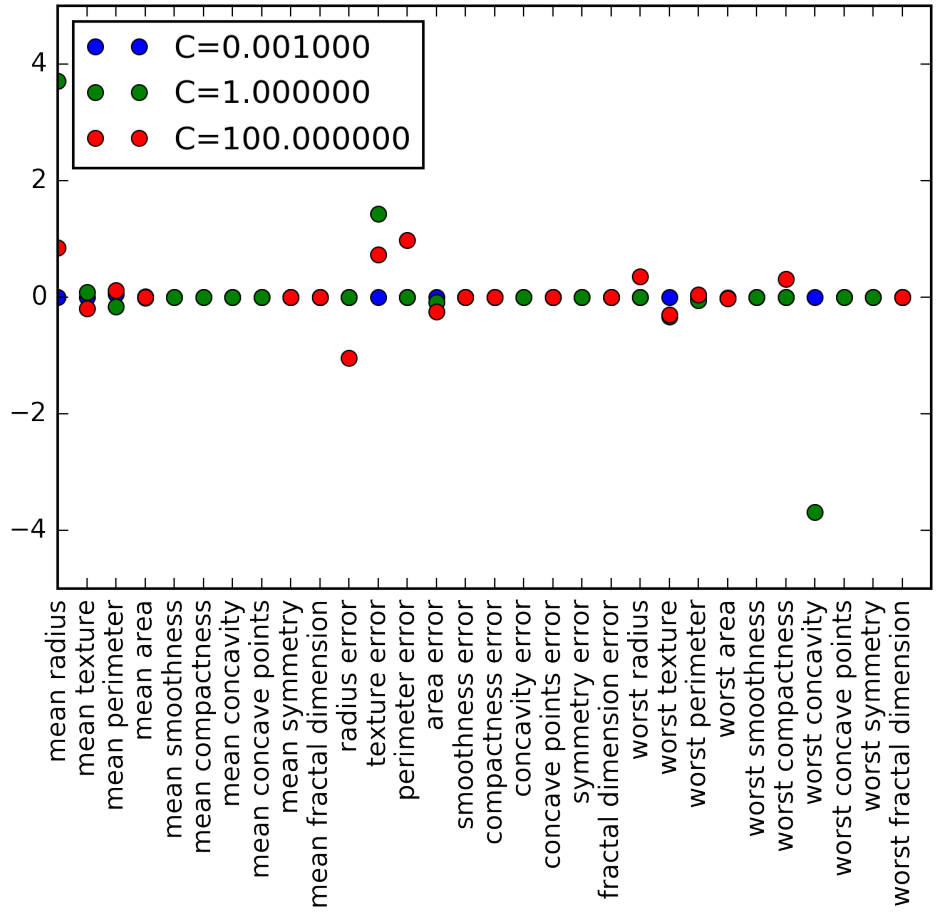


As `LogisticRegression` applies an L2 regularization by default, the result looks similar to `Ridge` in Figure ridge_coefficients. Stronger regularization pushes coefficients more and more towards zero, though coefficients never become exactly zero.

Inspecting the plot more closely, we can also see an interesting effect in the third coefficient, for "mean perimeter". For C=100 and C=1, the coefficient is negative, while for C=0.001, the coefficient is positive, with a magnitude that is even larger as for C=1. Interpreting a model like this, one might think the coefficient tells us which class a feature is associated with. For example, one might think that a high "texture error" feature is related to a sample being "malignant". However, the change of sign in the coefficient for "mean perimeter" means that depending on which model we look at, high "mean perimeter" could be either taken as being indicative of "benign" or indicative of "malignant". This illustrates that interpretations of coefficients of linear models should always be taken with a grain of salt.

If we desire a more interpretable model, using L1 regularization might help, as it limits the model to only using a few features. Here is the coefficient plot and classification accuracies for L1 regularization:

```
for C in [0.001, 1, 100]:
    lr_l1 = LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)
    print("training accuracy of L1 logreg with C=%f: %f"
          % (C, lr_l1.score(X_train, y_train)))
    print("test accuracy of L1 logreg with C=%f: %f"
          % (C, lr_l1.score(X_test, y_test)))
    plt.plot(lr_l1.coef_.T, 'o', label="C=%f" % C)

plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)

plt.ylim(-5, 5)
plt.legend(loc=2)
```



```
training accuracy of L1 logreg with C=0.001000: 0.913146

test accuracy of L1 logreg with C=0.001000: 0.923077

training accuracy of L1 logreg with C=1.000000: 0.960094

test accuracy of L1 logreg with C=1.000000: 0.958042

training accuracy of L1 logreg with C=100.000000: 0.985915

test accuracy of L1 logreg with C=100.000000: 0.979021
```

Linear Models for multiclass classification

Many linear classification models are binary models, and don't extend naturally to the multi-class case (with the exception of Logistic regression). A common technique to extend a binary classification algorithm to a multi-class classification algorithm is the *one-vs-rest* approach. In the one-vs-rest approach, a binary model is learned for each class, which tries to separate this class from all of the other classes, resulting in as many binary models as there are classes.

To make a prediction, all binary classifiers are run on a test point. The classifier that has the highest score on its single class "wins" and this class label is returned as prediction.

Having one binary classifier per class results in having one vector of coefficients w and one intercept b for each class. The class for which the result of formula

\begin{align*}

& w[0] *x[0]* + *w[1]* x[1] + \dotsc + w[p] * x[p] + b & \text{ (3) classification confidence}

\end{align*}

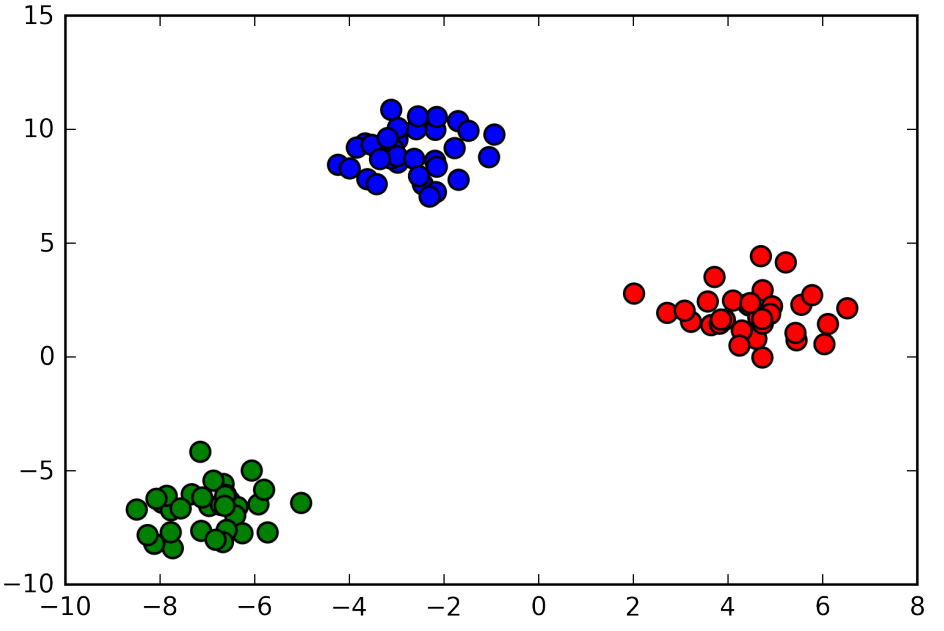
is highest is the assigned class label.

The mathematics behind logistic regression are somewhat different from the one-vs-rest approach, but they also result in one coefficient vector and intercept per class, and the same method of making a prediction is applied.

Let's apply the one-vs-rest method to a simple three-class classification dataset.

We use a two-dimensional dataset, where each class is given by data sampled from a Gaussian distribution.

```
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=42)
plt.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm3)
```



Now, we train a LinearSVC classifier on the dataset.

```
linear_svm = LinearSVC().fit(X, y)
print(linear_svm.coef_.shape)
print(linear_svm.intercept_.shape)
```

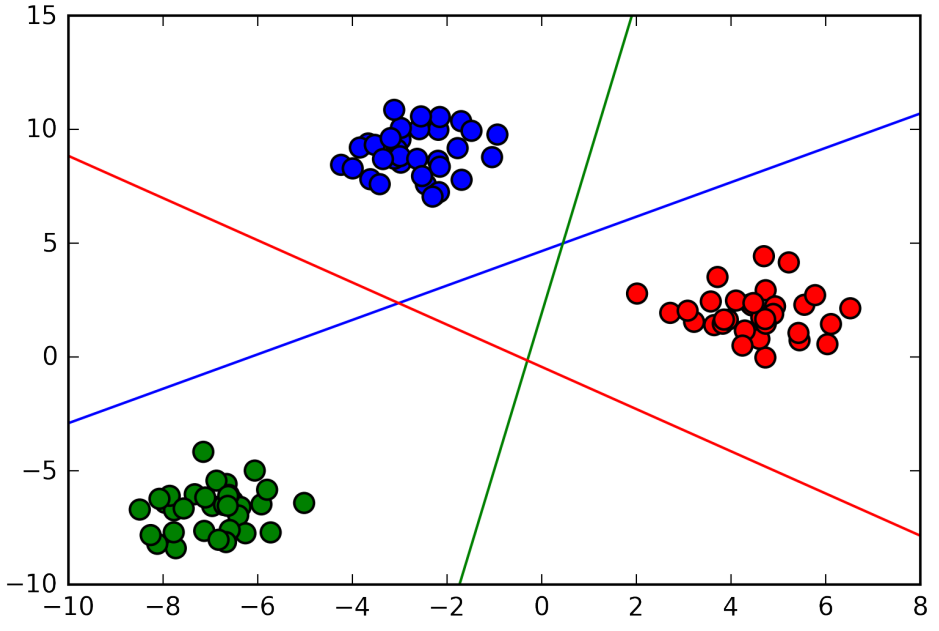
```
(3, 2)

(3,)
```

We see that the shape of the `coef_` is `(3, 2)`, meaning that each row of `coef_` contains the coefficient vector for one of the three classes. Each row has two entries, corresponding to the two features in the dataset.

The `intercept_` is now a one-dimensional array, storing the intercepts for each class.

Let's visualize the lines given by the three binary classifiers:

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm3)
line = np.linspace(-15, 15)
for coef, intercept in zip(linear_svm.coef_, linear_svm.intercept_):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1])
plt.ylim(-10, 15)
plt.xlim(-10, 8)
```



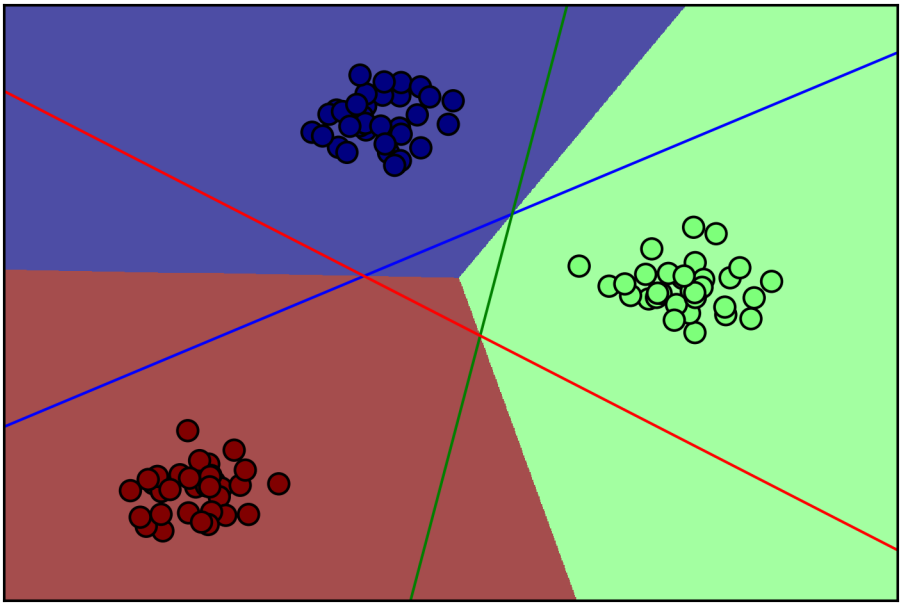
The red line shows the decision boundary for the binary classifier for the red class, and so on.

You can see that all the red points in the training data are below the red line, which means they are on the "red" side of this binary classifier. The red points are left of the green line, which means they are classified as "rest" by the binary classifier for the green class. The red points are below the blue line, which means the binary classifier for the blue class also classifies them as "rest". Therefore, any point in this area will be classified as red by the final classifier (Formula (3) of the red classifier is greater than zero, while it is smaller than zero for the other two classes).

But what about the triangle in the middle of the plot? All three binary classifiers classify points there as "rest". Which class would a point there be assigned to? The answer is the one with the highest value in Formula (3): the class of the closest line.

The following figure shows the prediction shown for all regions of the 2d space:

```
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
plt.scatter(X[:, 0], X[:, 1], c=y, s=60)
line = np.linspace(-15, 15)
for coef, intercept in zip(linear_svm.coef_, linear_svm.intercept_):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1])
```



Strengths, weaknesses and parameters

The main parameter of linear models is the regularization parameter, called `alpha` in the regression models and `C` in `LinearSVC` and `LogisticRegression`. Large `alpha` or small `C` mean simple models. In particular for the regression models, tuning this parameter is quite important. Usually `C` and `alpha` are searched for on a logarithmic scale.

The other decision you have to make is whether you want to use L1 regularization or L2 regularization. If you assume that only few of your features are actually important, you should use L1. Otherwise, you should default to L2.

L1 can also be useful if interpretability of the model is important. As L1 will use only a few features, it is easier to explain which features are important to the model, and what the effect of these features is.

Linear models are very fast to train, and also fast to predict. They scale to very large datasets and work well with sparse data. If your data consists of hundreds of thousands or millions of samples, you might want to investigate `SGDClassifier` and `SGDRegressor`, which implement even more scalable versions of the linear models described above.

Another strength of linear models is that they make i] relatively easy to understand how a prediction is made, using Formula (1) for regression and Formula (2) for clas-

sification. Unfortunately, it is often not entirely clear why coefficients are the way they are. This is particularly true if your dataset has highly correlated features; in these cases, the coefficients might be hard to interpret.

Linear models often perform well when the number of features is large compared to the number of samples. They are also often used on very large datasets, simply because other models are not feasible to train. However, on smaller dataset, other models might yield better generalization performance.

Naive Bayes Classifiers

Naive Bayes classifiers are a family of classifiers that are quite similar to the linear models discussed above. However, they tend to be even faster in training. The price paid for this efficiency is that naive Bayes models often provide generalization performance that is slightly worse than linear classifiers like `LogisticRegression` and `LinearSVC`.

The reason that naive Bayes models are so efficient is that they learn parameters by looking at each feature individually, and collect simple per-class statistics from each feature.

There are three kinds of naive Bayes classifiers implemented in scikit-learn, `GaussianNB`, `BernoulliNB` and `MultinomialNB`.

`GaussianNB` can be applied to any continuous data, while `BernoulliNB` assumes binary data and `MultinomialNB` assumes count data (that is each feature represents an integer count of something, like how often a word appears in a sentence). `BernoulliNB` and `MultinomialNB` are mostly used in text data classification, and we will revisit them in Chapter 7 (Text Data).

The `BernoulliNB` classifier counts how often every feature of each class is not zero. This is most easily understood with an example:

```
X = np.array([[0, 1, 0, 1],
              [1, 0, 1, 1],
              [0, 0, 0, 1],
              [1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])
```

Here, we have four data points, with four binary features each. There are two classes, 0 and 1.

For class 0 (the first and third data point), the first feature is zero 2 times and non-zero 0 times, the second features is zero 1 time and non-zero 1 time, and so on.

These same counts are then calculated for the data points in the second class.

Counting the non-zero entries per class in essence looks like this:

```
counts = {}
for label in np.unique(y):
    # iterate over each class
    # count (sum) entries of 1 per feature
    counts[label] = X[y == label].sum(axis=0)
print(counts)
```

```
{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

The other two naive Bayes models, `MultinomialNB` and `GaussianNB` are slightly different in what kind of statistics they compute. `MultinomialNB` takes into account the average value of each feature for each class, while `GaussianNB` stores the average value as well as the standard deviation of each feature for each class.

To make a prediction, a data point is compared to the statistics for each of the classes, and the best matching class is predicted. Interestingly, for `MultinomialNB` and `BernoulliNB`, this leads to a prediction formula that is of the same form as in the linear models (Formula (2)). Unfortunately, `coef_` for the naive Bayes models has a slightly different meaning then in the linear models, in that `coef_` is not the same as `w`.

Strengths, weaknesses and parameters

The `MultinomialNB` and `BernoulliNB` have a single parameter `alpha`, which controls model complexity. The way `alpha` works is that the algorithm adds `alpha` many virtual data points to the data, that have positive values for all the features. This results in a "smoothing" of the statistics. A large alpha means more smoothing, resulting in less complex models. The algorithms performance is relatively robust to the setting of alpha, meaning that setting `alpha` is not critical for good performance However, tuning it usually improves accuracy somewhat.

The `GaussianNB` model seems to be rarely used by practitioners, while the other two variants of naive Bayes are widely used for sparse count data such as text. MultinomialNB usually performs better than BinaryNB, in particular on datasets with a relatively large number of non-zero features (i.e. large documents).

The naive Bayes models share many of the strengths and weaknesses of the linear models. They are very fast to train and to predict, and the training procedure is easy to understand. The models work very well with high-dimensional sparse data, and are relatively robust to the parameters. Naive Bayes models are great baseline models, and are often used on very large datasets, where training even a linear model might take too long.

Decision trees

Decision trees are a widely used models for classification and regression tasks.

Essentially, they learn a hierarchy of "if-else" questions, leading to a decision.

These questions are similar to the questions you might ask in a game of twenty questions.

Imagine you want to distinguish between the following four animals: bears, hawks, penguins and dolphins.

Your goal is to get to the right answer b] asking as few if-else questions as possible.

You might start off by asking whether the animal has feathers, a question that narrows down your possible animals to just two animals.

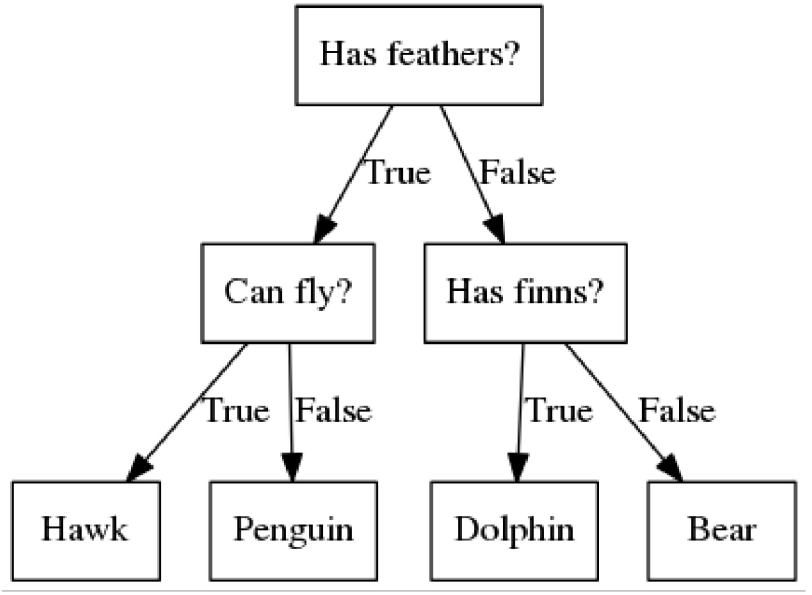
If the answer is yes, you can ask another question that could help you distinguish between hawks and penguins. For example, you could ask whether or not the animal can fly. If the animal doesn't have feathers, your possible animal choices are dolphins and bears, and you will need to ask a question to distinguish between these two animals, for example, asking whether the animal has fins.

This series of questions can be expressed as a decision tree, as shown in Figure animal_tree.

In this illustration, each node in the tree either represents a question, or a terminal node (also called a *leaf*) which contains the answer. The edges connect the answers to a question with the next question you would ask.

```
mglearn.plots.plot_animal_tree()
plt.suptitle("animal_tree");
```

animal_tree



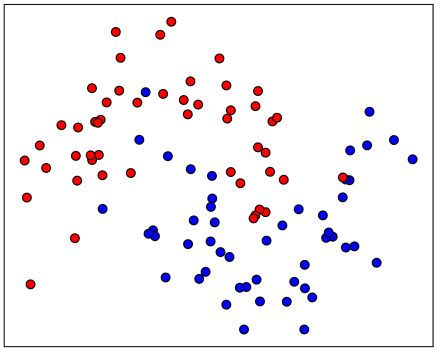
In machine learning parlance, we built a model to distinguish between four classes of animals (hawks, penguins, dolphins and bears) using the three features "has feathers", "can fly" and "has fins".

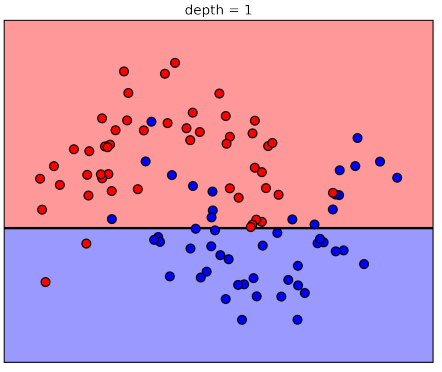
Instead of building these models by hand, we can learn them from data using supervised learning.

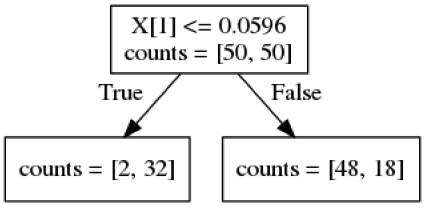
Building Decision Trees

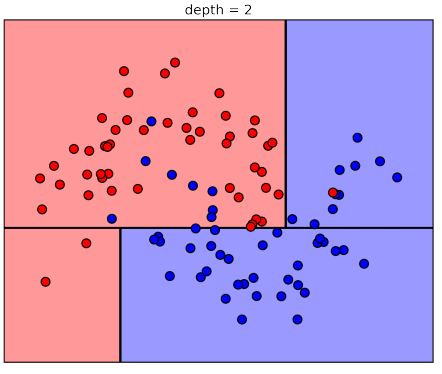
Let's go through the process of building a decision tree for the 2d classification dataset shown at the top of Figure tree_building. The dataset consists of two half-moon shapes of blue and red points, consisting of 75 data points each. We will refer to this dataset as `two_moons`.

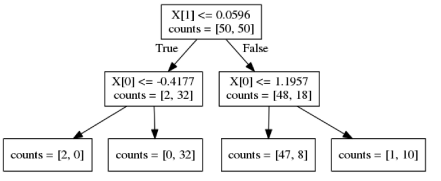
```
mglearn.plots.plot_tree_progressive()
plt.suptitle("tree_building");
```

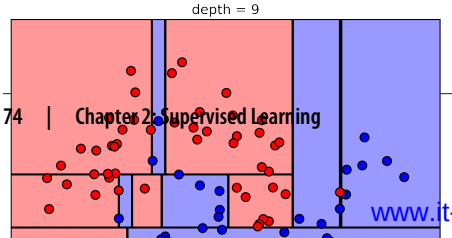


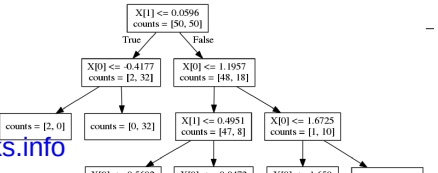












Learning a decision tree means learning a sequence of if/else questions that gets us to the true answer most quickly.

In the machine learning setting, these questions are called *tests* (not to be confused with the test set, which is the data we use to test to see how generalizable our model is).

Usually data does not come in the form of binary yes/no features as in the animal example, but is instead represented as continuous features such as in the 2d dataset shown in the figure. The tests that are used on continuous data are of the from "is feature i larger than value a".

To build a tree, the algorithm searches over all possible tests, and finds the one that is most informative about the target variable.

The second row in Figure tree_building shows the first test that is picked. Splitting the dataset vertically at `x[1]=0.2372` yields the most information; it best separates the blue points from the red points. The top node, also called the *root*, represents the whole dataset, consisting of 75 red and 75 blue points. The split is done by testing whether `x[1] <= 0.2372`, indicated by a black line. If the test is true, a point is assigned to the left node, which contains 8 blue points and 58 red points. Otherwise the point is assigned to the right node, which contains 67 red points and 17 blue points. These two nodes correspond to the top and bottom region shown in Figure tree_building.

Even though the first split did a good job of separating the blue and red points, the bottom region still contains blue points, and the top region still contains red points.

We can build a more accurate model by repeating the process of looking for the best test in both regions.

Figure tree_building shows that the most informative next split for the left and the right region are based on `x[0]`.

This recursive process yields a binary tree of decisions, with each node containing a test.

Alternatively, you can think of each test as splitting the part of the data that is currently considered along one axis. This yields a view of the algorithm as building a hierarchical partition. As each test concerns only a single feature, the regions in the resulting partition always have axis-parallel boundaries.

Figure tree_building illustrates the partitioning of the data in the left hand column, and the resulting tree in the right hand column.

The recursive partitioning of the data is usually repeated until each region in the partition (each leaf in the decision tree) only contains a single target value (a single class

or a single regression value). A leaf of the tree containing only one target value is called *pure*.

A prediction on a new data point is made by checking which region of the partition of the feature space the point lies in, and then predicting the majority target (or the single target in the case of pure leaves) in that region. The region can be found by traversing the tree from the root and going left or right, depending on whether the test is fulfilled or not.

Controlling complexity of Decision Trees

Typically, building a tree as described above, and continuing until all leaves are pure leads to models that are very complex and highly overfit to the training data. The presence of pure leaves mean that a tree is 100% accurate on the training set; each data point in the training set is in a leaf that has the correct majority class. The overfitting can be seen on the left of Figure tree_building in the bottom column. You can see the regions determined to be red in the middle of all the blue points. On the other hand, there is a small strip of blue around the single blue point to the very right. This is not how one would imagine the decision boundary to look, and the decision boundary focuses a lot on single outlier points that are far away from the other points in that class.

There are two common strategies to prevent overfitting: stopping the creation of the tree early, also called *pre-pruning*, or building the tree but then removing or collapsing nodes that contain little information, also called *post-pruning* or just *pruning*. Possible criteria for pre-pruning include limiting the maximum depth of the tree, limiting the maximum number of leaves, or requiring a minimum number of points in a node to keep splitting it.

Decision trees in scikit-learn are implemented in the `DecisionTreeRegressor` and `DecisionTreeClassifier` classes. Scikit-learn only implements *pre-pruning*, not *post-pruning*.

Let's look at the effect of pre-pruning in more detail on the breast cancer dataset.

As always, we import the dataset and split it into a training and test part.

Then we build a model using the default setting of fully developing the tree (growing the tree until all leaves are pure). We fix the `random_state` in the tree, which is used for tie-breaking internally.

```
from sklearn.tree import DecisionTreeClassifier

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
```

```
print("accuracy on training set: %f" % tree.score(X_train, y_train))
print("accuracy on test set: %f" % tree.score(X_test, y_test))
```

```
accuracy on training set: 1.000000

accuracy on test set: 0.937063
```

As expected, the accuracy on the training set is 100% as the leaves are pure.

The test-set accuracy is slightly worse than the linear models above, which had around 95% accuracy.

Now let's apply pre-pruning to the tree, which will stop developing the tree before we perfectly fit to the training data.

One possible way is to stop building the tree after a certain depth has been reached. Here we set `max_depth=4`, meaning only four consecutive questions can be asked (cf. Figure tree_building).

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

print("accuracy on training set: %f" % tree.score(X_train, y_train))
print("accuracy on test set: %f" % tree.score(X_test, y_test))
```

```
accuracy on training set: 0.988263

accuracy on test set: 0.951049
```

Limiting the depth of the tree decreases overfitting. This leads to a lower accuracy on the training set, but an improvement on the test set.

Analyzing Decision Trees

We can visualize the tree using the `export_graphviz` function from the tree module.

This writes a file in the `dot` file format, which is a text file format for storing graphs.

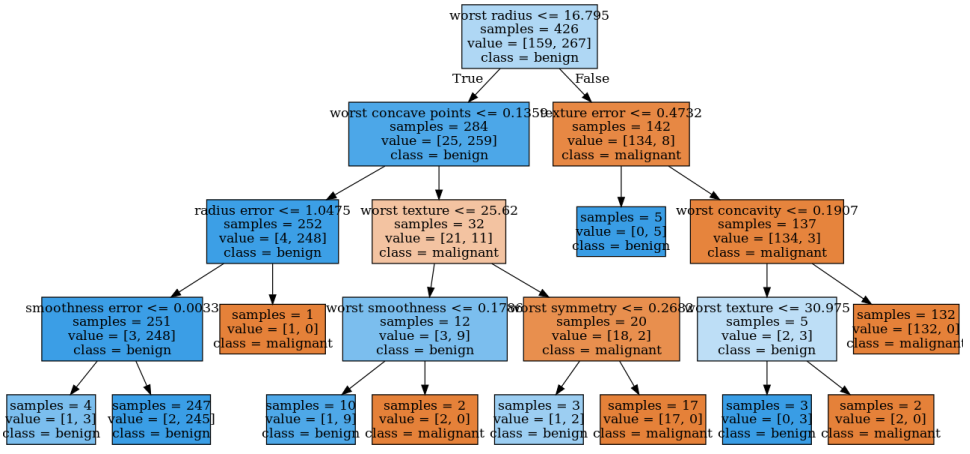
We set an option to color the nodes to reflect the majority class in each node and pass the class and features names so the tree can be properly labeled.

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="mytree.dot", class_names=["malignant", "benign"],
                feature_names=cancer.feature_names, impurity=False, filled=True)
```

We can read this file and visualize it using the graphviz module (or you can use any program that can read `dot` files):

```
import graphviz

with open("mytree.dot") as f:
```

```
dot_graph = f.read()
graphviz.Source(dot_graph)
```



The visualization of the tree provides a great in-depth view of how the algorithm makes predictions, and is a good example of a machine learning algorithm that is easily explained to non-experts. However, even with a tree of depth four, as seen here, the tree can become a bit overwhelming. Deeper trees (depth ten is not uncommon) are even harder to grasp.

One method of inspecting the tree that may be helpful is to find out which path most of the data actually takes.

The `n_samples` shown in each node in the figure gives the number of samples in each node, while `value` provides the number of samples per class.

Following the branches to the right, we see that `texture_error <= 0.4732` creates a node that only contains 8 benign but 134 malignant samples. The rest of this side of the tree then uses some finer distinctions to split off these 8 remaining benign samples. Of the 142 samples that went to the right in the initial split, nearly all of them (132) end up in the leaf to the very right.

Taking a left at the root, for `texture_error > 0.4732`, we end up with 25 malignant and 259 benign samples.

Nearly all of the benign samples end up in the second leave from the right, with most of the other leaves only containing very few samples.

Feature Importance in trees

Instead of looking at the whole tree, which can be taxing, there are some useful statistics that we can derive properties that we can derive to summarize the workings of the tree. The most commonly used summary is *feature importance*, which rates how

important each feature is for the decision a tree makes. It is a number between 0 and 1 for each feature, where 0 means "not used at all" and 1 means "perfectly predicts the target".

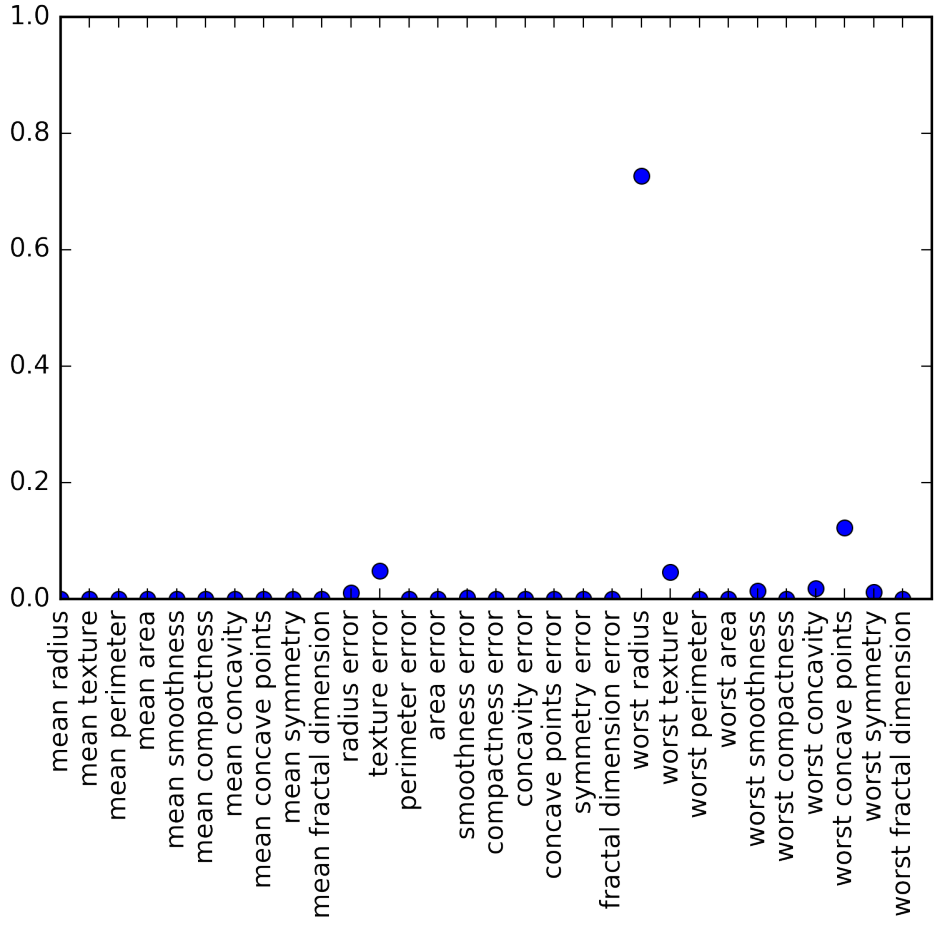
The feature importances always sum to one.

```
tree.feature_importances_
```

```
array([ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,

        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,

        0.01019737,  0.04839825,  0.        ,  0.        ,  0.0024156 ,

        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,

        0.72682851,  0.0458159 ,  0.        ,  0.        ,  0.0141577 ,

        0.        ,  0.018188  ,  0.1221132 ,  0.01188548,  0.        ])
```

We can visualize the feature importances in a way that is similar to the way we visualize the coefficients in the linear model.

```
plt.plot(tree.feature_importances_, 'o')
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)

plt.ylim(0, 1)
```

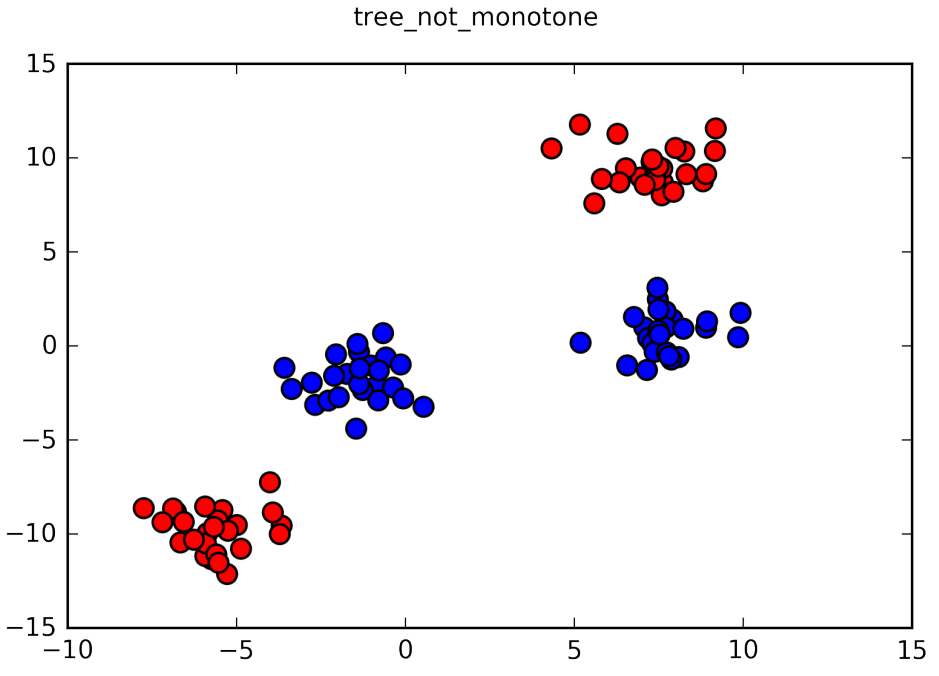


Here, we see that the feature used at the top split ("worst radius") is by far the most important feature. This confirms our observation in analyzing the tree, that the first level already separates the two classes fairly well.

However, if a feature has a low `feature_importance`, it doesn't mean that this feature is uninformative. It only means that this feature was not picked by the tree, likely because another feature encodes the same information.

In contrast to the coefficients in linear models, feature importances are always positive, and don't encode which class a feature is indicative of. The feature importances tell us that `worst radius` is important, but it does not tell us whether a high radius is indicative of a sample being "benign" or "malignant". In fact, there might not be such a simple relationship between features and class, as you can see in the example below:

```
tree = mglearn.plots.plot_tree_not_monotone()
plt.suptitle("tree_not_monotone")

tree
```




```
Feature importances: [ 0.  1.]
```

The plot shows a dataset with two features and two classes. Here, all the information is contained in `X[1]`, and `X[0]` is not used at all. But the relation between `X[1]` and the output class is not monotonous, meaning we cannot say "a high value of X[0] means class red, and a low value means class blue" or the other way around.

While we focuses our discussion here on decision trees for classification, all that was said is similarly true for decision trees for regression, as implemented in `DecisionTreeRegressor`. Both the usage and the analysis of regression trees are very similar to classification trees, so we won't go into any more detail here.

Strengths, weaknesses and parameters

As discussed above, the parameters that control model complexity in decision trees are the pre-pruning parameters that stop the building of the tree before it is fully developed. Usually picking one of the pre-pruning strategies, either setting `min_depth`, `max_leaf_nodes` or `min_samples_leaf` is to prevent overfitting.

Decision trees have two advantages over many of the algorithms we discussed so far: The resulting model can easily be visualized and understood by non-experts (at least for smaller trees), and the algorithms is completely invariant to scaling of the data: As each feature is processed separately, and the possible splits of the data don't depend on scaling, no preprocessing like normalization or standardization of features is needed for decision tree algorithms.

In particular, decision trees work well when you have features that are on completely different scales, or a mix of binary and continuous features.

The main down-side of decision trees is that even with the use of pre-pruning, decision trees tend to overfit, and provide poor generalization performance. Therefore, in most applications, the ensemble methods we discuss below are usually used in place of a single decision tree.

Ensembles of Decision Trees

Ensembles are methods that combine multiple machine learning models to create more powerful models.

There are many models in the machine learning literature that belong to this category, but there are two ensemble models that have proven to be effective on a wide range of datasets for classification and regression, both of which use decision trees as their building block: Random Forests and Gradient Boosted Decision Trees.

Random Forests

As observed above, a main drawback of decision trees is that they tend to overfit the training data. Random forests

are one way to address this problem. Random forests are essentially a collection of decision trees, where each tree is slightly different from the others.

The idea of random forests is that each tree might do a relatively good job of predicting, but will likely overfit on part of the data.

If we build many trees, all of which work well and overfit in different ways, we can reduce the amount of overfitting by averaging their results. This reduction in overfitting, while retaining the predictive power of the trees, can be shown using rigorous mathematics.

To implement this strategy, we need to build many decision tree. Each tree should do an acceptable job of predicting the target, and should also be different from the other trees. Random forests get their name from injecting randomness into the tree building to ensure each tree is different. There are two ways in which the trees in a random

forest are randomized: by selecting the data points used to build a tree and by selecting the features in each split test. Let's go into this process in more detail.

Building Random Forests

To build a random forest model, you need to decide on the number of trees to build (the `n_estimator` parameter of `RandomForestRegressor` or `RandomForestClassifier`). Lets say we want to build ten trees. These trees will be built completely independent from each other, and [will?] make random choices to make sure they are distinct [the trees make random choices?].

To build a tree, we first take what is called a *bootstrap* sample of our data. A bootstrap sample means from our `n_samples` data points, we repeatedly draw an example randomly with replacement (i.e. the same sample can be picked multiple times), `n_samples` times. This will create a dataset that is as big as the original dataset, but some data points will be missing from it, and some will be repeated.

To illustrate, lets say we want to create a bootstrap sample of the list `['a', 'b', 'c', 'd']`. A possible bootstrap sample would be `['b', 'd', 'd', 'c']`. Another possible sample would be `['d', 'a', 'd', 'a']`.

Next, a decision tree is built based on this newly created dataset. However, the algorithm we described for the decision tree is slightly modified. Instead of looking for the best test for each node, in each node the algorithm randomly selects a subset of the features, and looks for the best possible test involving one of these features. The amount of features that is selected is controlled by the `max_features` parameter.

This selection of a subset of features is repeated separately in each node, so that each node in a tree can make a decision using a different subset of the features.

The bootstrap sampling leads to each decision tree in the random forest being built on a slightly different dataset. Because of the selection of features in each node, each split in each tree operates on a different subset of features. Together these two mechanisms ensure that all the trees in the random forests are different.

A critical parameter in this process is `max_features`. If we set `max_features` to `n_features`, that means that each split can look at all features in the dataset, and no randomness will be injected. If we set `max_features` to one, that means that the splits have no choice at all on which feature to test, and can only search over different thresholds for the feature that was selected randomly.

Therefore, a high `max_features` means that the trees in the random forest will be quite similar, and they will be able to fit the data easily, using the most distinctive features. A low `max_features` means that the trees in the random forest will be quite different, and that each tree might need to be very deep in order to fit the data well.

To make a prediction using the random forest, the algorithm first makes a prediction for every tree in the forest. For regression, we can average these results to get our final prediction. For classification, a "soft voting" strategy is used. This means each algorithm makes a "soft" prediction, providing a probability for each possible output label. The probabilities predicted by all the trees are averaged, and the class with the highest label is predicted.

Analyzing Random Forests

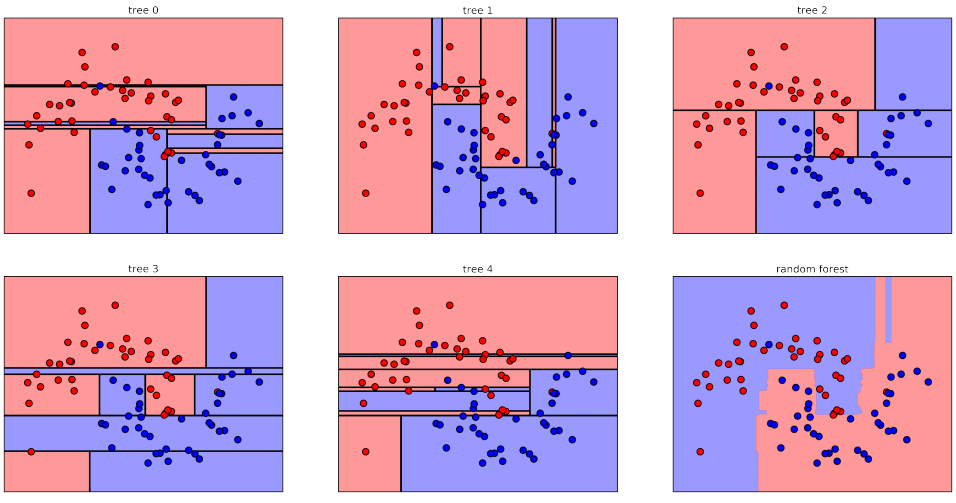
Let's apply a random forest consisting of five trees to the `two_moon` data we studied above.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)

forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',

            max_depth=None, max_features='auto', max_leaf_nodes=None,

            min_samples_leaf=1, min_samples_split=2,

            min_weight_fraction_leaf=0.0, n_estimators=5, n_jobs=1,

            oob_score=False, random_state=2, verbose=0, warm_start=False)
```

The trees that are built as part of the random forest are stored in the `estimator_` attribute. Let's visualize the decision boundaries learned by each tree, together with their aggregate prediction, as made by the forest.

```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("tree %d" % i)
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)
mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1], alpha=.4)
axes[-1, -1].set_title("random forest")
plt.scatter(X_train[:, 0], X_train[:, 1], c=np.array(['r', 'b'])[y_train], s=60)
```



You can clearly see that the decisions learned by the five trees are quite different. Each of them makes some mistakes, as some of the training points that are plotted here were not actually included in the training set of the tree, due to the bootstrap sampling.

The random forest overfit less than any of the trees individually, and provides a much more intuitive decision boundary. In any real application, we would use many more trees (often hundreds or thousands), leading to even smoother boundaries.

Let's apply a random forest consisting of 100 trees on the breast cancer dataset:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)

print("accuracy on training set: %f" % forest.score(X_train, y_train))
print("accuracy on test set: %f" % forest.score(X_test, y_test))
```

```
accuracy on training set: 1.000000

accuracy on test set: 0.972028
```

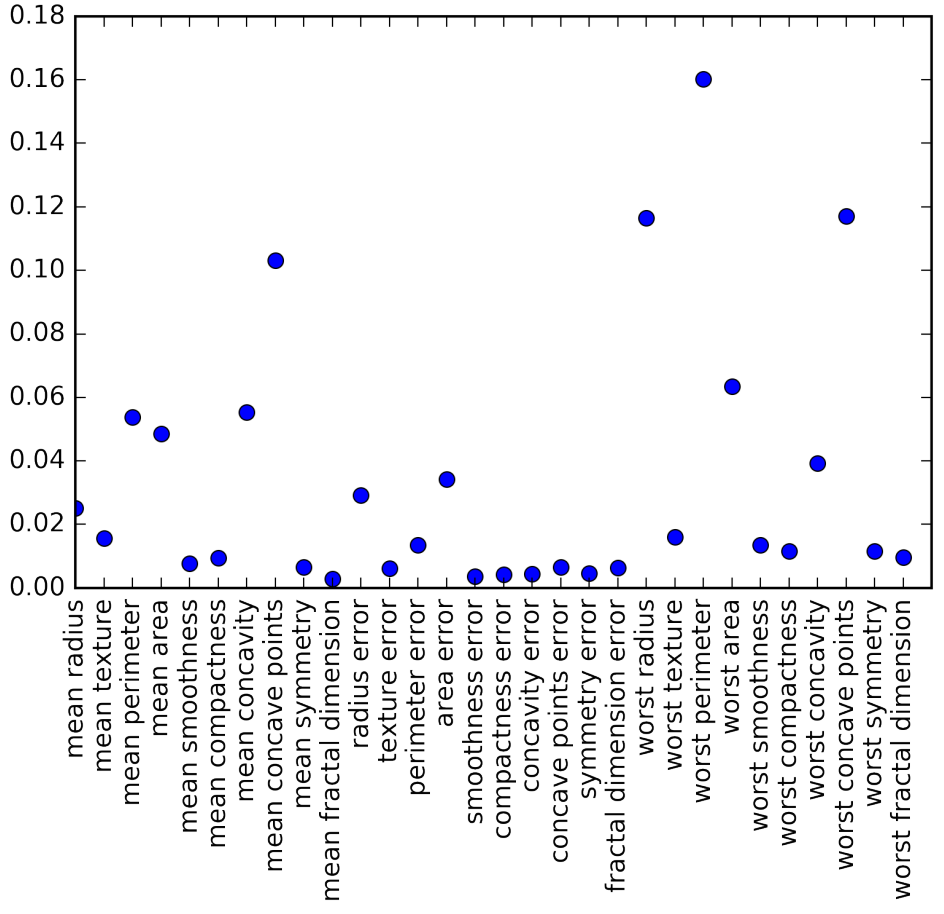
The random forest gives us an accuracy of 97%, better than the linear models or a single decision tree, without tuning any parameters. We could adjust the `max_features` setting, or apply pre-pruning as we did for the single decision tree.

However, often the default parameters of the random forest already work quite well.

Similarly to the decision tree, the random forest provides feature importances, which are computed by aggregating the feature importances over the trees in the forest. Typ-

ically the feature importances provided by the random forest are more reliable than the ones provided by a single tree.

```
plt.plot(forest.feature_importances_, 'o')
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90);
```



As you can see, the random forest gives non-zero importance to many more features than the single tree.Similarly to the single decision tree, the random forest also gives a lot of importance to the "worst radius", but it actually chooses "worst perimeter" to be the most informative feature overall. The randomness in building the random forest forces the algorithm to consider many possible explanations, the result of which being that the random forest captures a much broader picture of the data than a single tree.

Strengths, weaknesses and parameters

Random forests for regression and classification are currently among the most widely used machine learning methods.

They are very powerful, often work well without heavy tuning of the parameters, and don't require scaling of the data.

Essentially, random forests share all of the benefits of decision trees, while making up for some of their deficiencies.

One reason to still use decision trees is if you need a compact representation of the decision making process. It is basically impossible to interpret tens or hundreds of trees in detail, and trees in random forests tend to be deeper than decision trees (because of the use of feature subsets). Therefore, if you need to summarize the prediction making in a visual way to non-experts, a single decision tree might be a better choice.

While building random forests on large dataset might be somewhat time-consuming, it an be parallelized across multiple CPU cores within a computer easily. If you are using a multi-core processor (as nearly all modern computers do), you can use the `n_jobs` parameter to adjust the number of cores to use. Using more CPU cores will result in linear speed-ups (using two cores, the training of the random forest will be twice as fast), but specifying `n_jobs` larger than the number of cores will not help. You can set `n_jobs=-1` to use all the cores in your computer.

You should keep in mind that random forests, by their nature, are random, and setting different random states (or not setting the `random_state` at all) can drastically change the model that is built. The more trees there are in the forest, the more robust it will be against the choice of random state. If you want to have reproducible results, it is important to fix the `random_state`.

Random forests don't tend to perform well on very high dimensional, sparse data, such as text data. For this kind of data, linear models might be more appropriate.

Random forests usually work well even on very large datasets, and training can easily be parallelized over many CPU cores within a powerful computer. However, random forests require more memory and are slower to train and to predict than linear models. If time and memory are important in an application, it might make sense to use a linear model instead.

The important parameters to adjust are `n_estimators`, `max_features` and possibly pre-pruning options like `max_depth`. For `n_estimators`, larger is always better. Averaging more trees will yield a more robust ensemble. However, there are diminishing returns, and more trees need more memory and more time to train. A common rule of thumb is to build "as many as you have time / memory for".

As described above `max_features` determines how random each tree is, and a smaller `max_features` reduces overfitting. The default values, and a good rule of thumb, are `max_features=sqrt(n_features)` for classification and `max_features=log2(n_features)` for regression.

Adding `max_features` or `max_leaf_nodes` might sometimes improve performance. It can also drastically reduce space and time requirements for training and prediction.

Gradient Boosted Regression Trees (Gradient Boosting Machines)

Gradient boosted regression trees is another ensemble method that combines multiple decision trees to a more powerful model. Despite the "regression" in the name, these models can be used for regression and classification.

In contrast to random forests, gradient boosting works by building trees in a serial manner, where each tree tries to correct the mistakes of the previous one. There is no randomization in gradient boosted regression trees; instead, strong pre-pruning is used. Gradient boosted trees often use very shallow trees, of depth one to five, often making the model smaller in terms of memory, and making predictions faster.

The main idea behind gradient boosting is to combine many simple models (in this context known as *weak learners*), like shallow trees. Each tree can only provide good predictions on part of the data, and so more and more trees are added to iteratively improve performance.

Gradient boosted trees are frequently the winning entries in machine learning competitions, and are widely used in industry. They are generally a bit more sensitive to parameter settings than random forests, but can provide better accuracy if the parameter are set correctly.

Apart from the pre-pruning and the number of trees in the ensemble, another important parameter of gradient boosting is the `learning_rate` which controls how strongly each tree tries to correct the mistakes of the previous trees. A higher learning rate means each tree can make stronger corrections, allowing for more complex models. Similarly, adding more trees to the ensemble, which can be done by increasing `n_estimators`, also increases the model complexity, as the model has more chances to correct mistakes on the training set.

Here is an example of using `GradientBoostingClassifier` on the breast cancer dataset.

By default, 100 trees of maximum depth three are used, with a learning rate of 0.1.

```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
```

```
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)

print("accuracy on training set: %f" % gbrt.score(X_train, y_train))
print("accuracy on test set: %f" % gbrt.score(X_test, y_test))
```

```
accuracy on training set: 1.000000

accuracy on test set: 0.958042
```

As the training set accuracy is 100%, we are likely to be overfitting. To reduce overfitting, we could either apply stronger pre-pruning by limiting the maximum depth or lower the learning rate:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

print("accuracy on training set: %f" % gbrt.score(X_train, y_train))
print("accuracy on test set: %f" % gbrt.score(X_test, y_test))
```

```
accuracy on training set: 0.990610

accuracy on test set: 0.972028
```

```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)

print("accuracy on training set: %f" % gbrt.score(X_train, y_train))
print("accuracy on test set: %f" % gbrt.score(X_test, y_test))
```

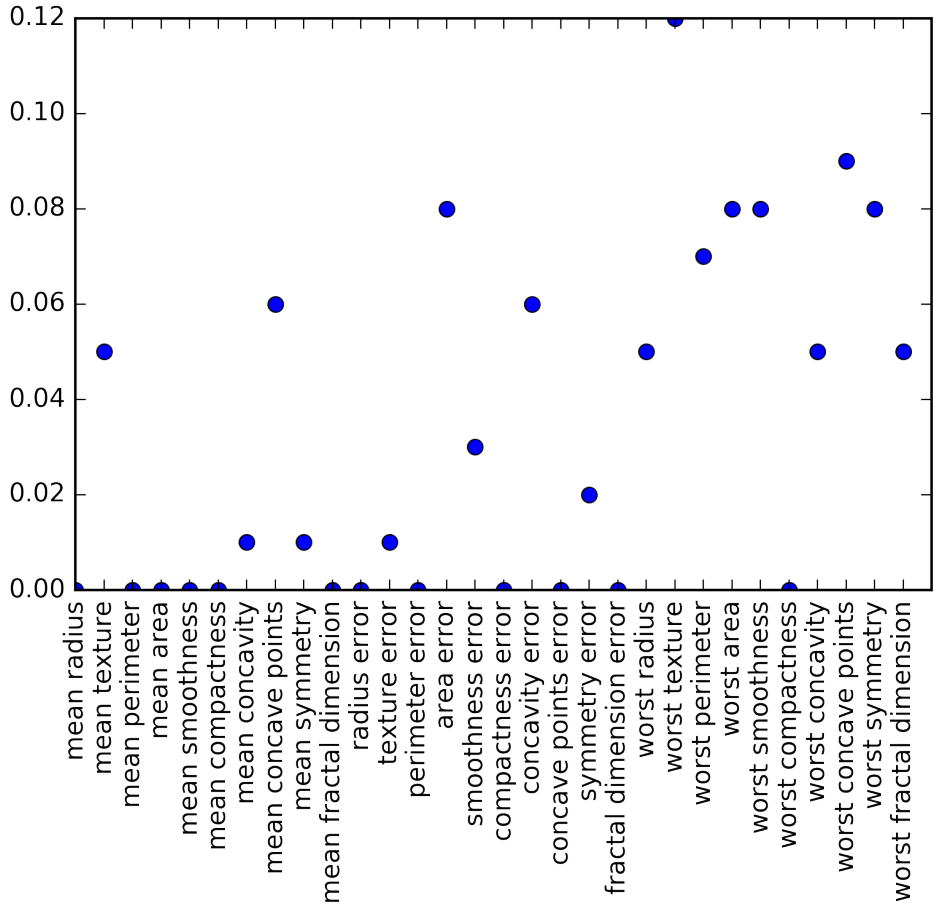
```
accuracy on training set: 0.988263

accuracy on test set: 0.965035
```

Both methods of decreasing the model complexity decreased the training set accuracy as expected. In this case, lowering the maximum depth of the trees provided a significant improvement of the model, while lowering the learning rate only

increased the generalization performance slightly.

As for the other decision tree based models, we can again visualize the feature importances to get more insight into our model. As we used 100 trees, it is impractical to inspect them all, even if they are all of depth 1.

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

plt.plot(gbrt.feature_importances_, 'o')
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90);
```



We can see that the feature importances of the gradient boosted trees are somewhat similar to the feature importances of the random forests, though the gradient boosting completely ignored some of the features.

As gradient boosting and random forest perform well on similar kinds of data, a common approach is to first try random forests, which work quite robustly. If random forests work well, but prediction time is at a premium, or it is important to squeeze out the last percentage of accuracy from the machine learning model, moving to gradient boosting often helps.

If you want to apply gradient boosting to a large scale problem, it might be worth looking into the `xgboost` package and its python interface, which at the time of writing is faster (and sometimes easier to tune) than the scikit-learn implementation of gradient boosting on many datasets.

Strengths, weaknesses and parameters

Gradient boosted decision trees are among the most powerful and widely used models for supervised learning.

Their main drawback is that they require careful tuning of the parameters, and may take a long time to train.

Similarly to other tree-based models, the algorithm works well without scaling and on a mixture of binary and continuous features. As other tree-based models, it also often does not work well on high-dimensional sparse data.

The main parameters of the gradient boosted tree models are the number of trees `n_estimators`, and the `learning_rate`, which controls how much each tree is allowed to correct the mistakes of the previous trees.

These two parameters are highly interconnected, as a lower `learning_rate` means that more trees are needed to build a model of similar complexity. In contrast to random forests, where higher `n_estimators` is always better, increasing `n_estimators` in gradient boosting leads to a more complex model, which may lead to overfitting.

A common practice is to fit `n_estimators` depending on the time and memory budget, and then search over different `learning_rates`.

Another important parameter is `max_depth`, which is usually very low for gradient boosted models, often not deeper than five splits.

Kernelized Support Vector Machines

The next type of supervised model we will discuss is kernelized support vector machines (SVMs).

We already saw linear support vector machines for classification in the linear model section. Kernelized support vector machines (often just referred to as SVMs) are an extension that allows for more complex models which are not defined simply by hyperplanes in the input space. While there are support vector machines for classification and regression, we will restrict ourself to the classification case, as implemented in `SVC`. Similar concepts apply to support vector regression, as implemented in `SVR`.

The math behind kernelized support vector machines is a bit involved, and is mostly beyond the scope of this book.

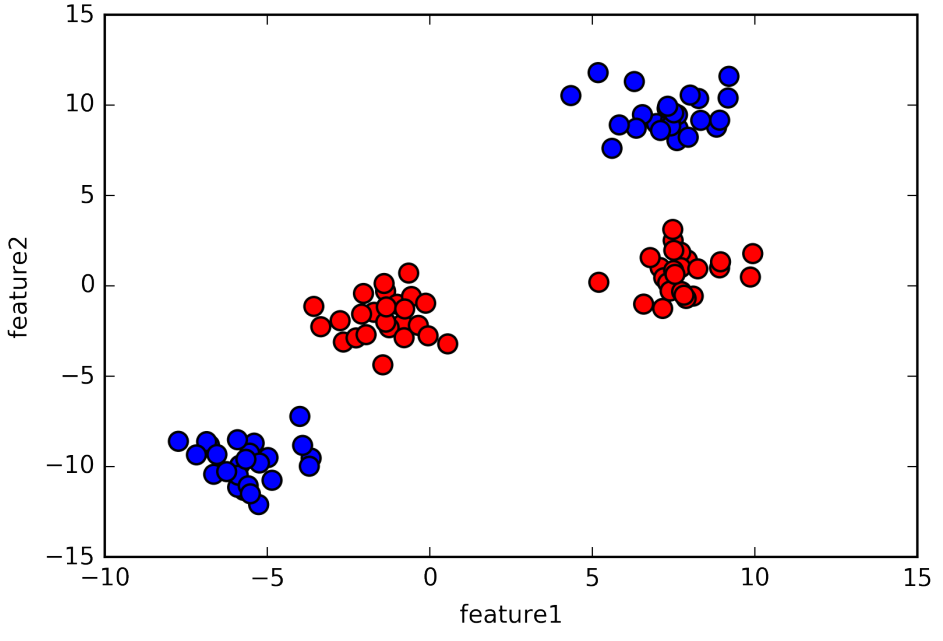
However, we will try to give you some intuitions about the idea behind the method.

Linear Models and Non-linear Features

As you saw in Figure linear_classifiers, linear models can be quite limiting in low-dimensional spaces, as lines or hyperplanes have limited flexibility. One way to make a linear model more flexible is by adding more features, for example by adding interactions or polynomials of the input features.

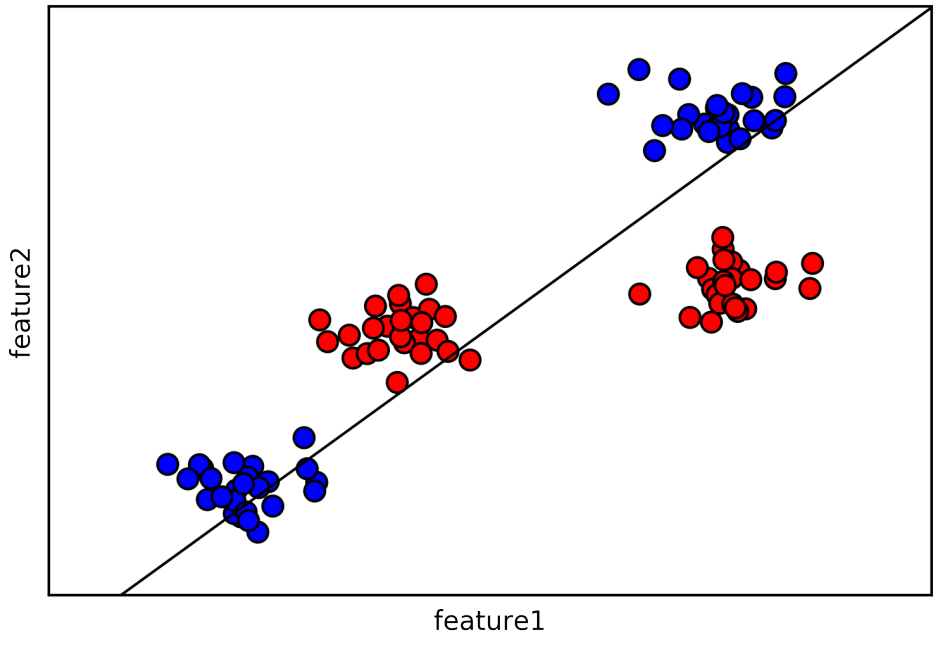
Let's look at the synthetic dataset we used in Figure tree_not_monotone:

```
X, y = make_blobs(centers=4, random_state=8)
y = y % 2

plt.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm2)
plt.xlabel("feature1")
plt.ylabel("feature2")
```



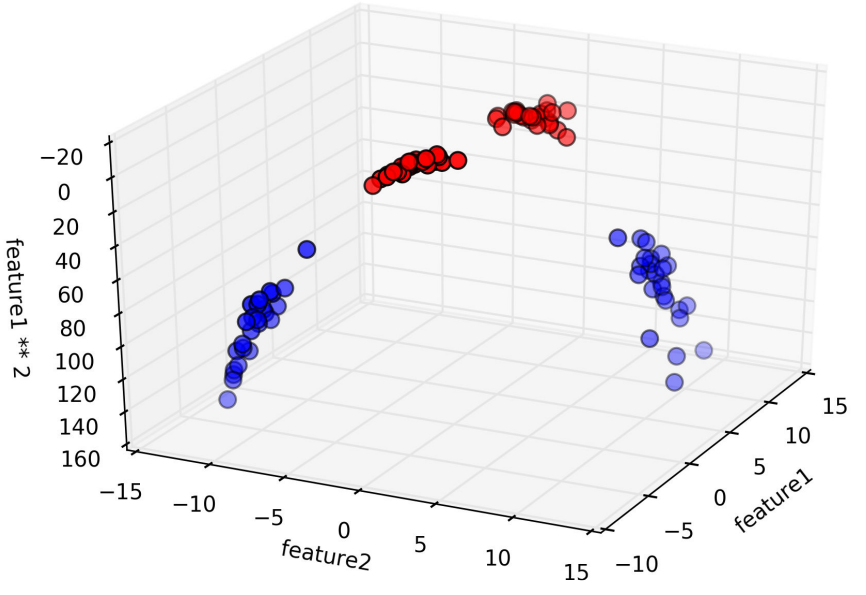
A linear model for classification can only separate points using a line, and will not be able to do a very good job on this dataset:

```
from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)

mglearn.plots.plot_2d_separator(linear_svm, X)
plt.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm2)
plt.xlabel("feature1")
plt.ylabel("feature2")
```



Now, let's expand the set of input features, say by also adding `feature2 ** 2`, the square of the second feature, as a new feature. Instead of representing each data point as a two-dimensional point (`feature1, feature2`), we now represent it as a three-dimensional point (`feature1, feature2, feature2 ** 2`) (Footnote: We picked this particular feature to add for illustration purposes. The choice is not particular important.). This new representation is illustrated below in a three-dimensional scatter plot:

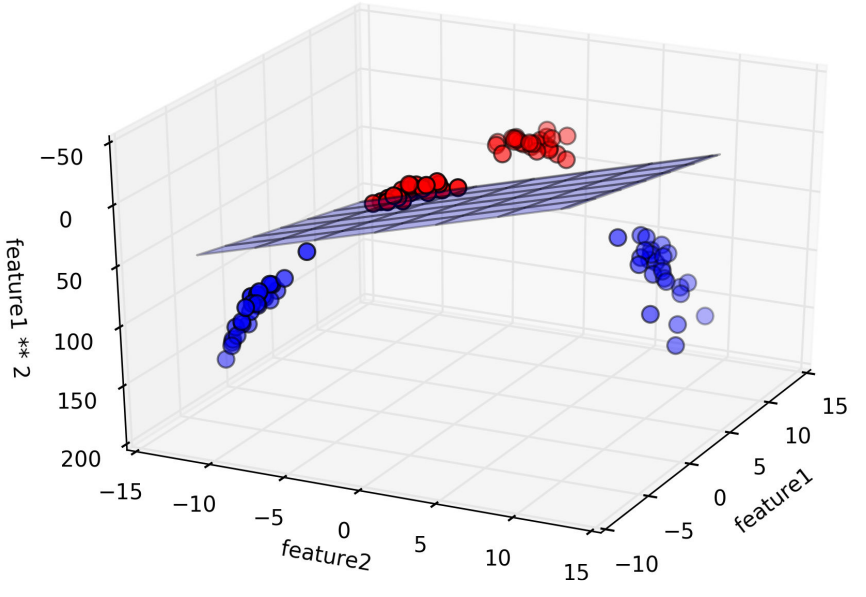
```
# add the squared first feature
X_new = np.hstack([X, X[:, 1:] ** 2])


from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
# visualize in 3D
ax = Axes3D(figure, elev=-152, azim=-26)
ax.scatter(X_new[:, 0], X_new[:, 1], X_new[:, 2], c=y, cmap=mglearn.cm2, s=60)
ax.set_xlabel("feature1")
ax.set_ylabel("feature2")
ax.set_zlabel("feature1 ** 2")
```



In the new, three-dimensional representation of the data, it is now indeed possible to separate the red and the blue points

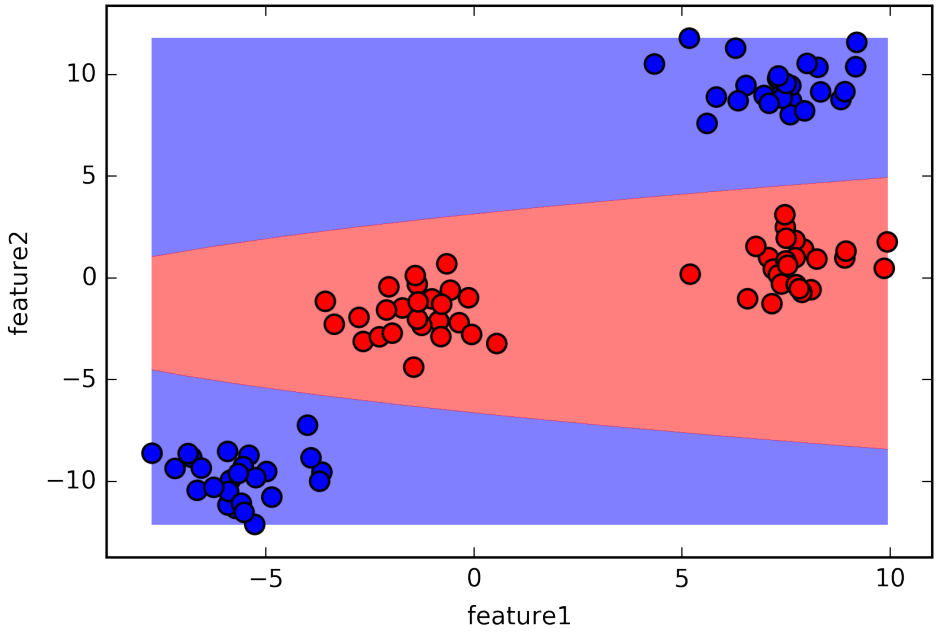
using a linear model, a plane in three dimensions. We can confirm this by fitting a linear model to the augmented data:

```
linear_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_

# show linear decision boundary
figure = plt.figure()
ax = Axes3D(figure, elev=-152, azim=-26)
xx = np.linspace(X_new[:, 0].min(), X_new[:, 0].max(), 50)
yy = np.linspace(X_new[:, 1].min(), X_new[:, 1].max(), 50)

XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
ax.scatter(X_new[:, 0], X_new[:, 1], X_new[:, 2], c=y, cmap=mglearn.cm2, s=60)
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)

ax.set_xlabel("feature1")
ax.set_ylabel("feature2")
ax.set_zlabel("feature1 ** 2")
```



As a function of the original features, the linear SVM model is not actually linear anymore. It is not a line, but more of an ellipse.

```
ZZ = YY ** 2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],
             cmap=mglearn.cm2, alpha=0.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm2)
plt.xlabel("feature1")
plt.ylabel("feature2")
```



The Kernel Trick

The lesson here is that adding non-linear features to the representation of our data can make linear models much more powerful. However, often we don't know which features to add, and adding many features (like all possible interactions in a 100 dimensional feature space) might make computation very expensive.

Luckily, there is a clever mathematical trick that allows us to learn a classifier in a higher dimensional space without actually computing the new, possibly very large representation. This trick is known as the *kernel trick*.

The kernel trick works by directly computing the distance (more precisely, the scalar products) of the data points for the expanded feature representation, without ever actually computing the expansion.

There are two ways to map your data into a higher dimensional space that are commonly used with support vector machines: the polynomial kernel, which computes all possible polynomials up to a certain degree of the original features (like `feature1 ** 2 * feature2 ** 5`), and the radial basis function (rbf) kernel, also known as Gaussian kernel.

The Gaussian kernel is a bit harder to explain, as it corresponds to an infinite dimensional feature space. One way to explain the Gaussian kernel is that it considers all possible polynomials of all degrees, but the importance of the features decreases for

higher degrees. [Footnote: this follows from the Taylor expansion of the exponential map].

If all of this is too much math talk for you, don't worry. You can still use SVMs without trying to imagine infinite dimensional feature spaces. In practice, how a SVM with an rbf kernel makes a decision can be summarized quite easily.

Understanding SVMs

During training, the SVM learns how important each of the training data points is to represent the decision boundary between the two classes. Typically only a subset of the training points matter for defining the decision boundary: the ones that lie on the border between the classes. These are called *support vectors* and give the support vector machine its name.

To make a prediction for a new point, the distance to the support vectors is measured. A classification decision is made based on the distance to the support vectors, and the importance of the support vectors that was learned during training (stored in the `dual_coef_` attribute of `SVC`).

The way distance between data points is measured by the Gaussian kernel:

\begin{align*}

&k_\text{rbf}(x_1, x_2) = \exp(\gamma||x_1 - x_2||^2) &\text{ (4) Gaussian kernel}

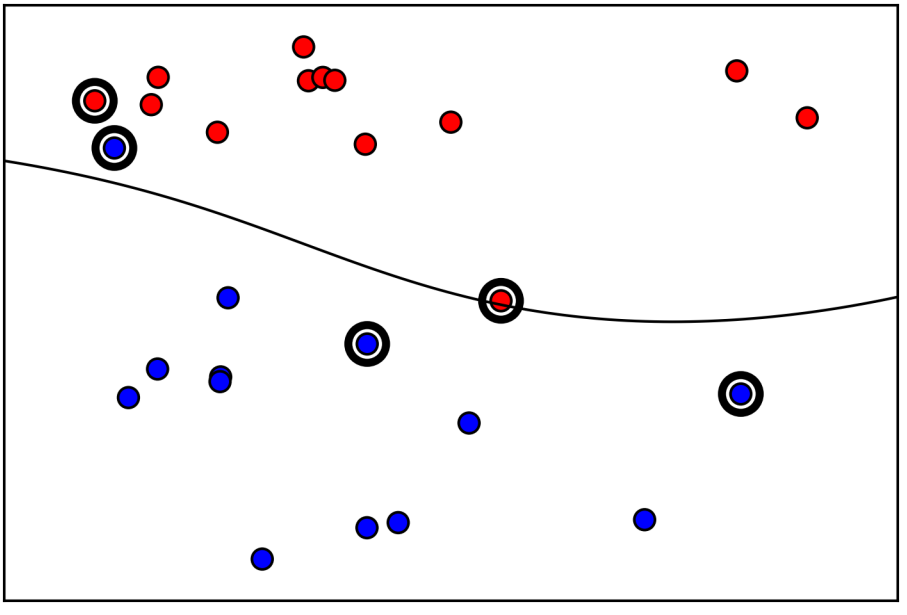
\end{align*}

Here, x_1 and x_2 are data points, $||x_1 - x_2 ||$ denotes Euclidean distance and γ is a parameter that controls the width of the Gaussian kernel.

Below is the result of training an support vector machine on a two-dimensional two-class dataset.

The decision boundary is shown in black, and the support vectors are the points with wide black circles.

```
from sklearn.svm import SVC

X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
# plot data
plt.scatter(X[:, 0], X[:, 1], s=60, c=y, cmap=mglearn.cm2)
# plot support vectors
sv = svm.support_vectors_
plt.scatter(sv[:, 0], sv[:, 1], s=200, facecolors='none', zorder=10, linewidth=3)
```



In this case, the SVM yields a very smooth and non-linear (not a straight line) boundary.

There are two parameters we adjusted here: The `C` parameter and the `gamma` parameter, which we will now discuss in detail.

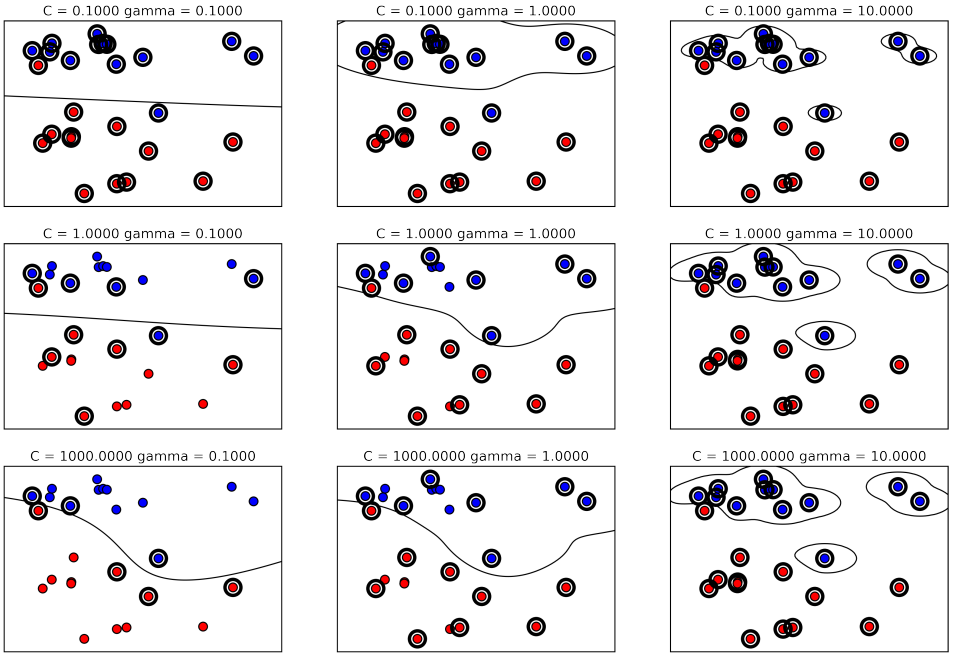
Tuning SVM parameters

The `gamma` parameter is the one shown in Formula (4), which controls the width of the Gaussian kernel. It determines the scale of what it means for points to be close together.

The `C` parameter is a regularization parameter similar to the linear models. It limits the importance of each point (or more precisely, their `dual_coef_`).

Let's have a look at what happens when we vary these parameters:

```
fig, axes = plt.subplots(3, 3, figsize=(15, 10))

for ax, C in zip(axes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):
        mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)
```



Going from left to right, we increase the parameter `gamma` from 0.1 to 10. A small gamma means a large radius for the Gaussian kernel, which means that many points are considered close-by. This is reflected in very smooth decision boundaries on the left, and boundaries that focus more on single points further to the right. A low value of gamma means that the decision boundary will vary slowly, which yields a model of low complexity, while a high value of gamma yields a more complex model.

Going from top to bottom, we increase the `C` parameter from 0.1 to 1000. As with the linear models, a small C means a very restricted model, where each data point can only have very limited influence. You can see that in the top left, the decision boundary looks nearly linear, with the red and blue points that are misclassified barely changing the line.

Increasing `C`, as shown on the bottom right, allows these points to have a stronger influence on the model, and makes the decision boundary bend to correctly classify them.

Let's apply the rbf kernel SVM to the breast cancer dataset. By default, `C=1` and `gamma=1./n_features`.

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

svc = SVC()
svc.fit(X_train, y_train)
```

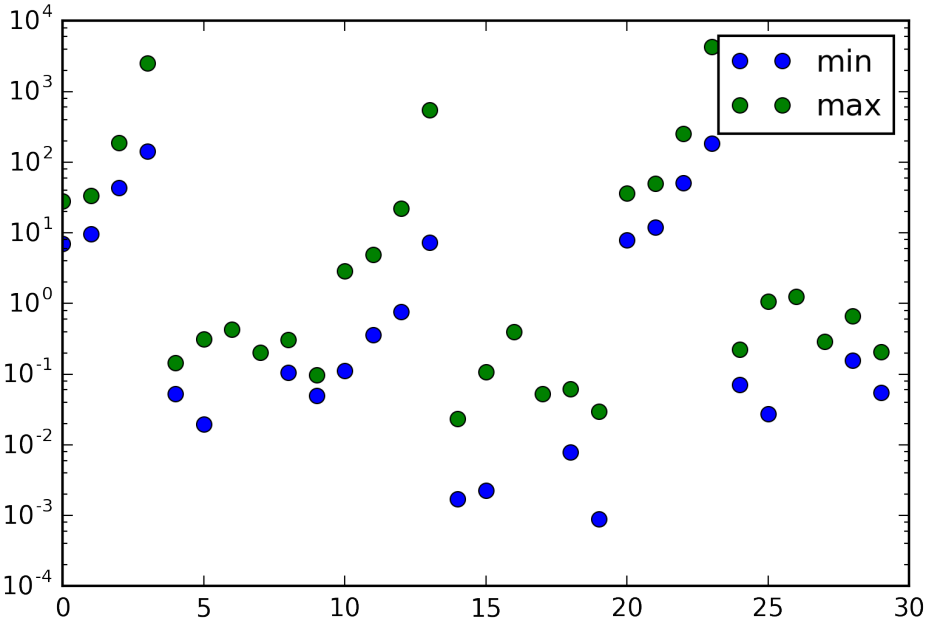
```
print("accuracy on training set: %f" % svc.score(X_train, y_train))
print("accuracy on test set: %f" % svc.score(X_test, y_test))
```

```
accuracy on training set: 1.000000

accuracy on test set: 0.629371
```

The model overfit quite substantially, with a perfect score on the training set and only 62% accuracy on the test set.

While SVMs often perform quite well, they are very sensitive to the settings of the parameters, and to the scaling of the data. In particular, they require all the features to vary on a similar scale. Let's look at the minimum and maximum values for each feature, plotted in log-space:

```
plt.plot(X_train.min(axis=0), 'o', label="min")
plt.plot(X_train.max(axis=0), 'o', label="max")
plt.legend(loc="best")
plt.yscale("log")
```



From this plot we can determine that features in the breast cancer dataset are of completely different orders of magnitude.

This can be somewhat of a problem for other models (like linear models), but it has devastating effects for the kernel SVM.

Preprocessing Data for SVMs

One way to resolve this problem is by rescaling each feature, so that they are approximately on the same scale.

A common rescaling methods for kernel SVMs is to scale the data such that all features are between zero and one. We will see how to do this using the `MinMaxScaler` preprocessing method in Chapter 3 (Unsupervised Learning), where we'll give more details.

For now, let's do this "by hand":

```
# Compute the minimum value per feature on the training set
min_on_training = X_train.min(axis=0)
# Compute the range of each feature (max - min) on the training set
range_on_training = (X_train - min_on_training).max(axis=0)

# subtract the min, divide by range
# afterwards min=0 and max=1 for each feature
X_train_scaled = (X_train - min_on_training) / range_on_training
print("Minimum for each feature\n%s" % X_train_scaled.min(axis=0))
print("Maximum for each feature\n %s" % X_train_scaled.max(axis=0))
```

```
Minimum for each feature

[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.

  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]

Maximum for each feature

 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.

  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

```
# use THE SAME transformation on the test set,
# using min and range of the training set. See Chapter 3 (unsupervised learning) for details.
X_test_scaled = (X_test - min_on_training) / range_on_training

svc = SVC()
svc.fit(X_train_scaled, y_train)

print("accuracy on training set: %f" % svc.score(X_train_scaled, y_train))
print("accuracy on test set: %f" % svc.score(X_test_scaled, y_test))
```

```
accuracy on training set: 0.948357

accuracy on test set: 0.951049
```

Scaling the data made a huge difference! Now we are actually in an underfitting regime, where training and test set performance are quite similar. From here, we can try increasing either `C` or `gamma` to fit a more complex model:

```
svc = SVC(C=1000)
svc.fit(X_train_scaled, y_train)

print("accuracy on training set: %f" % svc.score(X_train_scaled, y_train))
print("accuracy on test set: %f" % svc.score(X_test_scaled, y_test))
```

```
accuracy on training set: 0.988263

accuracy on test set: 0.972028
```

Here, increasing `C` allows us to improve the model significantly, resulting in 97.2% accuracy.

Strengths, weaknesses and parameters

Kernelized support vector machines are very powerful models and perform very well on a variety of datasets.

SVMs allow for very complex decision boundaries, even if the data has only a few features. SVMs work well on low-dimensional and high-dimensional data (i.e. few and many features), but don't scale very well with the number of samples. Running on data with up to 10000 samples might work well, but working with datasets of size 100000 or more can become challenging in terms of runtime and memory usage.

Another downside of SVMs is that they require careful preprocessing of the data and tuning of the parameters.

For this reason, SVMs have been replaced by tree-based models such as random forests (that require little or no preprocessing) in many applications. Furthermore, SVM models are hard to inspect; it can be difficult to understand why a particular prediction was made, and it might be tricky to explain the model to a non-expert.

Still it might be worth trying SVMs, particularly if all of your features represent measurements in similar units (i.e. all are pixel intensities) and are on similar scales.

The important parameters in kernel SVMs are the regularization parameter `C`, the choice of the kernel, and the kernel-specific parameters. We only talked about the `rbf` kernel in any depth above, but other choices are available in scikit-learn. The `rbf` kernel has only one parameter, `gamma`, which is the inverse of the width of the Gaussian kernel. `gamma` and `C` both control the complexity of the model, with large values in either resulting in a more complex model. Therefore, good settings for the two parameters are usually strongly correlated, and `C` and `gamma` should be adjusted together.

Neural Networks (Deep Learning)

A family of algorithms known as neural networks has recently seen a revival under the name "deep learning".

While deep learning shows great promise in many machine learning applications, many deep learning algorithms are tailored very carefully to a specific use-case. Here, we will only discuss some relatively simple methods, namely *multilayer perceptrons* for classification and regression, that can serve as a starting point for more involved deep learning methods. Multilayer perceptrons (MLPs) are also known as (vanilla) feed-forward neural networks, or sometimes just neural networks.

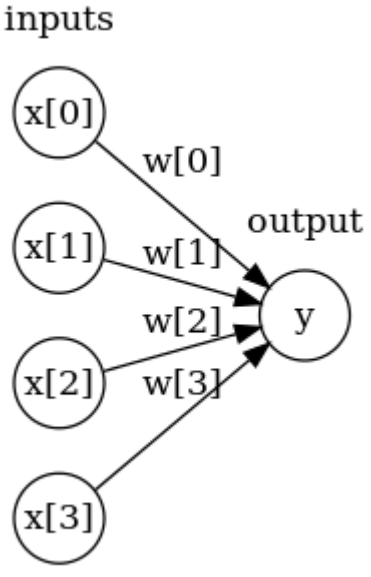
The Neural Network Model

MLPs can be viewed as generalizations of linear models which perform multiple stages of processing to come to a decision.

Remember that the prediction by a linear regressor is given as:

In words, `y` is a weighted sum of the input features `x[0]` to `x[p]`, weighted by the learned coefficients `w[0]` to `w[p]`. We could visualize this graphically as:

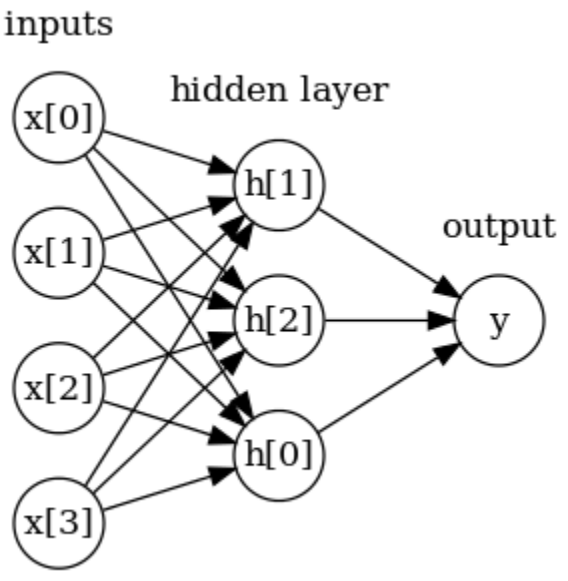
```
mglearn.plots.plot_logistic_regression_graph()
```



where each node on the left represents an input feature, the connecting lines represent the learned coefficients, and the node on the right represents the output, which is a weighted sum of the inputs.

In an MLP, this process of computing weighted sums is repeated multiple times, first computing *hidden units* that represent an intermediate processing step, which are again combined using weighted sums, to yield the final result:

```
print("Figure single_hidden_layer")
mglearn.plots.plot_single_hidden_layer_graph()
```



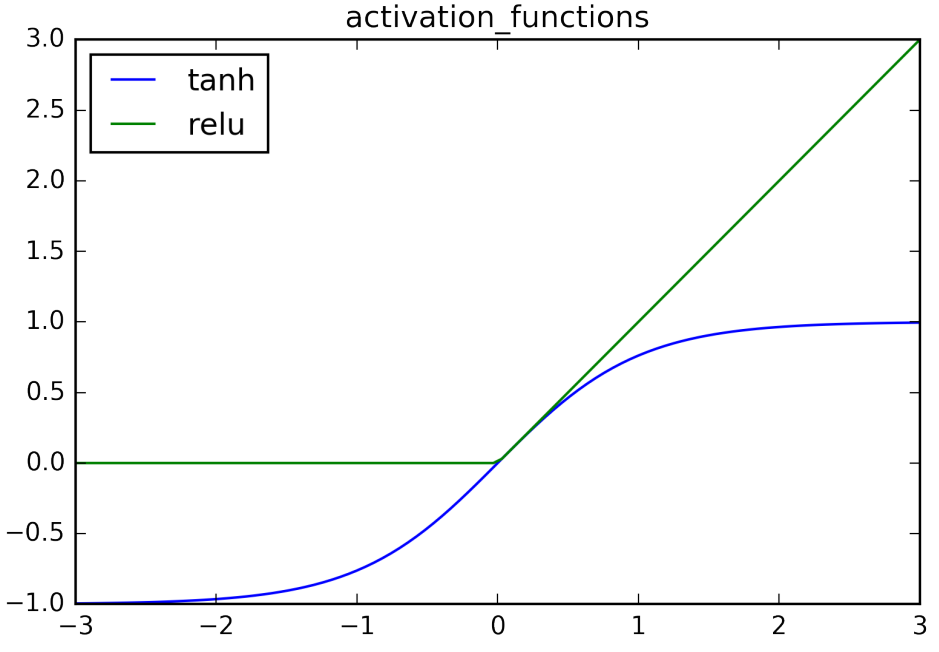

```
Figure single_hidden_layer
```

This model has a lot more coefficients (also called weights) to learn: there is one between every input and every hidden unit (which make up the *hidden layer*), and one between every unit in the hidden layer and the output.

Computing a series of weighted sums is mathematically the same as computing just one weighted sum, so to make this model truly more powerful than a linear model, there is one extra trick we need. After computing a weighted sum for each hidden unit, a non-linear function is applied to the result, usually the *rectifying nonlinearity* (also known as rectified linear unit or relu) or the *tangens hyperbolicus* (tanh). The result of this function is then used in the weighted sum that computes the output `y`.

The two functions are visualized in Figure activation_functions. The relu cuts off values below zero, while tanh saturates to -1 for low input values and +1 for high input values. Either non-linear function allows the neural network to learn much more complicated function than a linear model could.

```
line = np.linspace(-3, 3, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.legend(loc="best")
plt.title("activation_functions")
```



For the small neural network pictures in Figure single_hidden_layer above, the full formula for computing y in the case of regression would be (when using a tanh nonlinearity):

Here, `w` are the weights between the input `x` and the hidden layer `h`, and `v` are the weights between the hidden layer `h` and the output `y`. The weights `v` and `w` are learned from data, `x` are the input features, `y` is the computed output, and `h` are intermediate computations.

An important parameter that needs to be set by the user is the number of nodes in the hidden layer, and can be as small as 10 for very small or simple datasets, and can be as big as 10000 for very complex data.

It is also possible add additional hidden layers, as in Figure two_hidden_layers below. Having large neural networks made up of many of these layers of computation is what inspired the term "deep learning".

```
print("Figure two_hidden_layers")
mglearn.plots.plot_two_hidden_layer_graph()
```

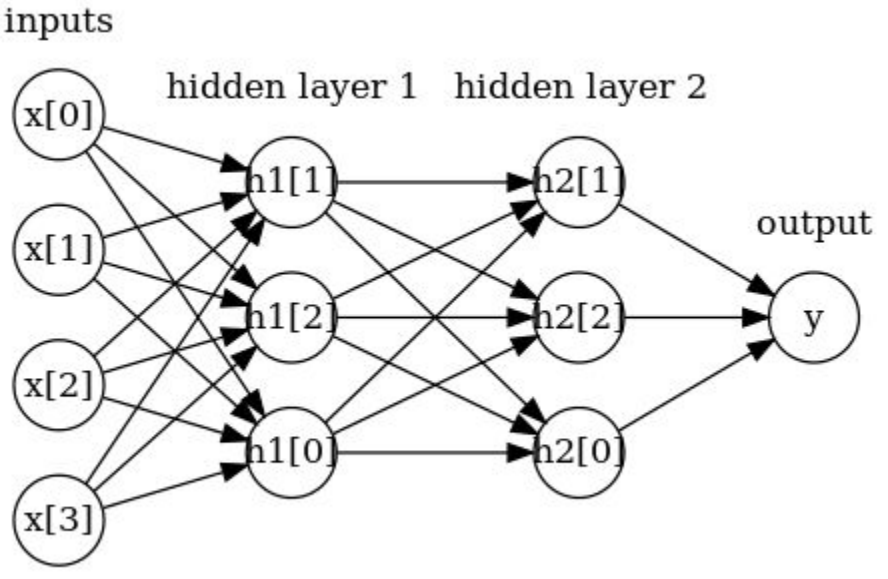
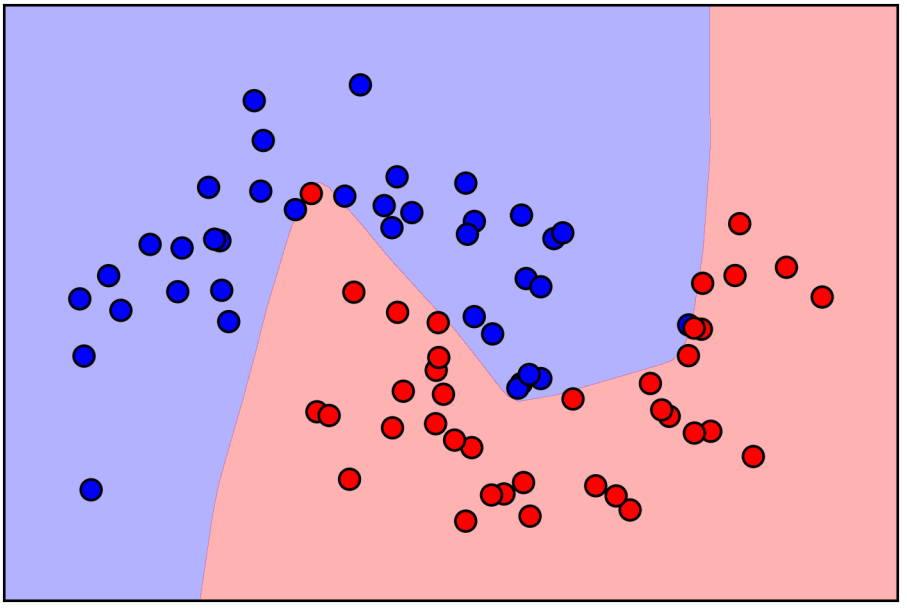


Figure two_hidden_layers

Tuning Neural Networks

Let's look into the workings of the MLP by applying the `MLPClassifier` to the `two_moons` dataset we saw above.

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)


mlp = MLPClassifier(algorithm='l-bfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=60, cmap=mglearn.cm2)
```

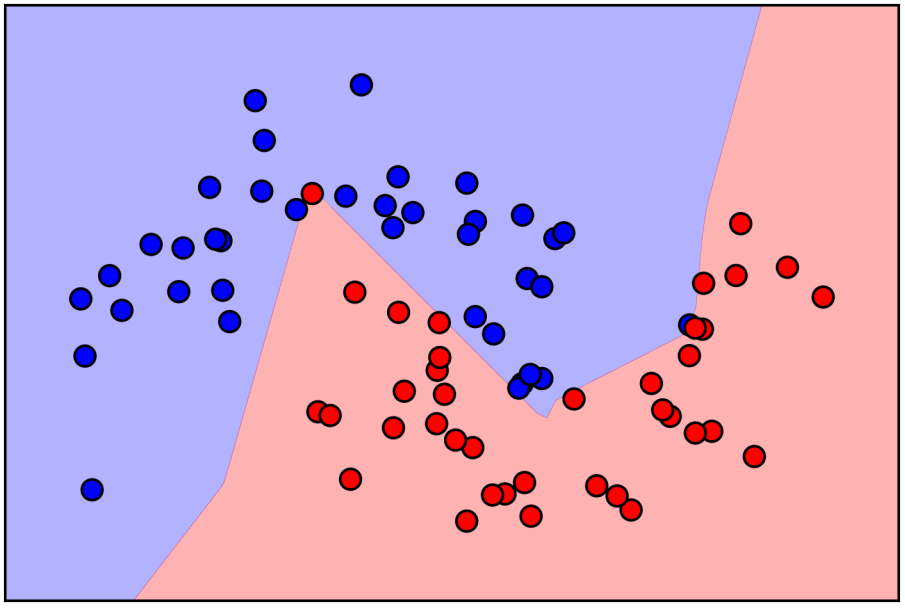


As you can see, the neural network learned a very nonlinear but relatively smooth decision boundary.

We used `algorithm='l-bfgs'` which we will discuss later.

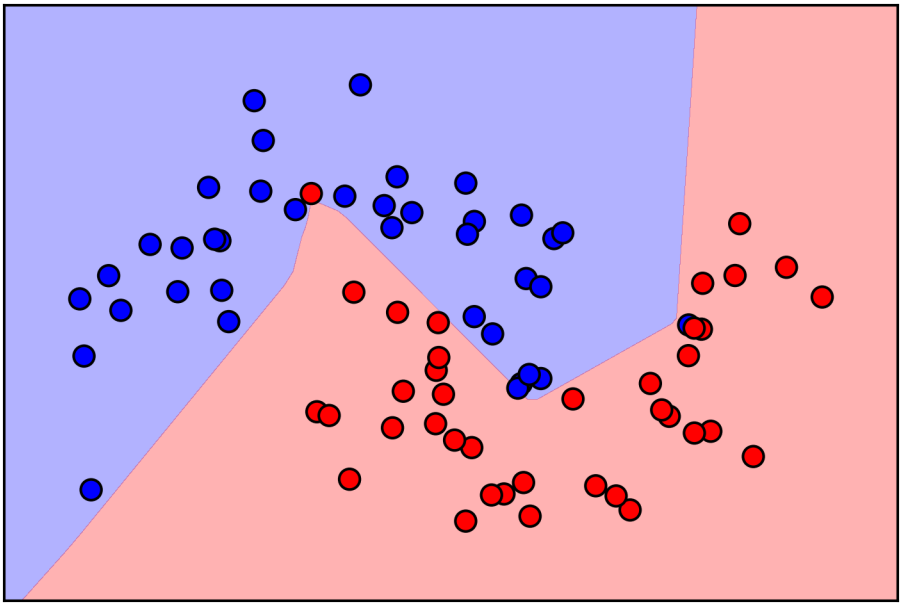
By default, the MLP uses 100 hidden nodes, which is quite a lot for this small dataset. We can reduce the number (which reduces the complexity of the model) and still get a good result:

```
mlp = MLPClassifier(algorithm='l-bfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=60, cmap=mglearn.cm2)
```

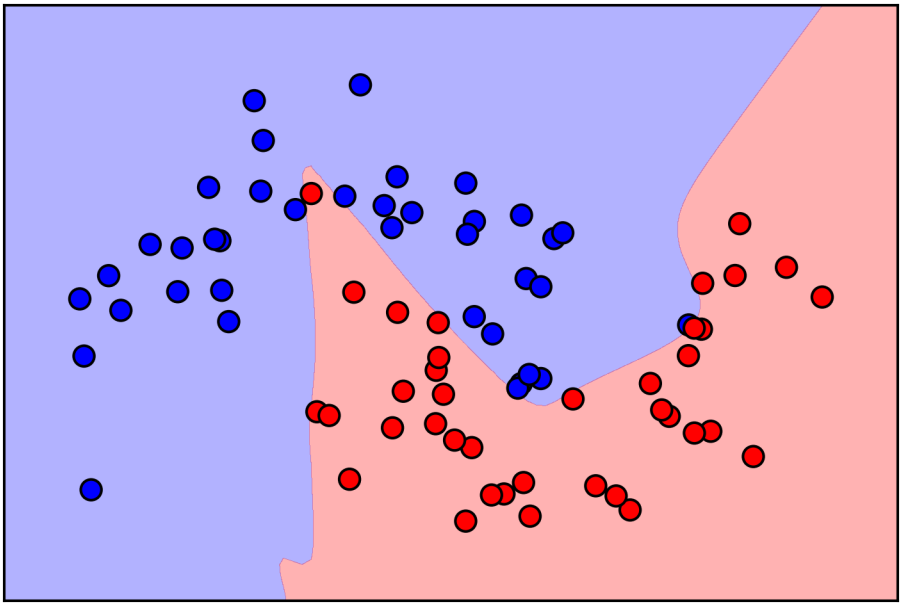


With only 10 hidden units, the decision boundary looks somewhat more ragged. The default nonlinearity is ‘relu', shown in Figure activation_function. With a single hidden layer, this means the decision function will be made up of 10 straight line segments. If we want a smoother decision boundary, we could either add more hidden units (as in the figure above), add second hidden layer, or use the “tanh” nonlinearity:

```
# using two hidden layers, with 10 units each
mlp = MLPClassifier(algorithm='l-bfgs', random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=60, cmap=mglearn.cm2)
```



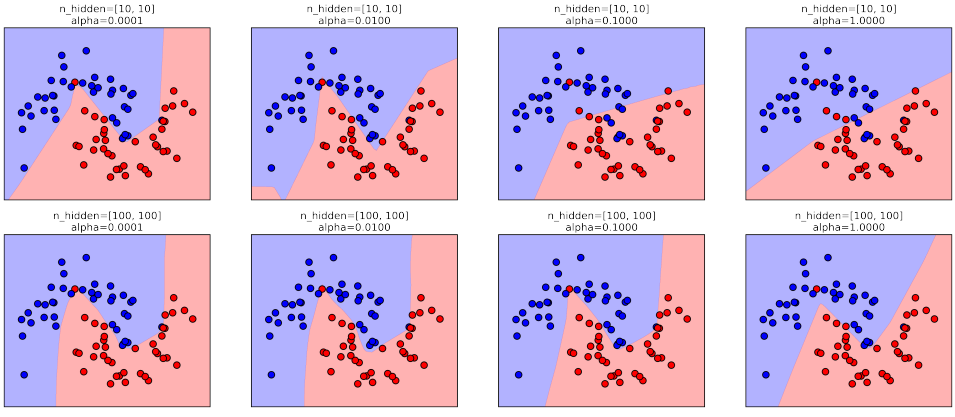
```
# using two hidden layers, with 10 units each, now with tanh nonlinearity.
mlp = MLPClassifier(algorithm='l-bfgs', activation='tanh',
                    random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=60, cmap=mglearn.cm2)
```



Finally, we can also control the complexity of a neural network by using an "l2" penalty to shrink the weights towards zero, as we did in ridge regression and the linear classifiers. The parameter for this in the `MLPClassifier` is `alpha` (as in the linear regression models), and is set to a very low value (little regularization) by default.

Here is the effect of different values of `alpha` on the `two_moons` dataset, using two hidden layers of 10 or 100 units each:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for ax, n_hidden_nodes in zip(axes, [10, 100]):
    for axx, alpha in zip(ax, [0.0001, 0.01, 0.1, 1]):
        mlp = MLPClassifier(algorithm='l-bfgs', random_state=0,
                            hidden_layer_sizes=[n_hidden_nodes, n_hidden_nodes],
                            alpha=alpha)
        mlp.fit(X_train, y_train)
        mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=axx)
        axx.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=60, cmap=mglearn.cm2)
        axx.set_title("n_hidden=[%d, %d]\nalpha=%.4f"
                      % (n_hidden_nodes, n_hidden_nodes, alpha))
```



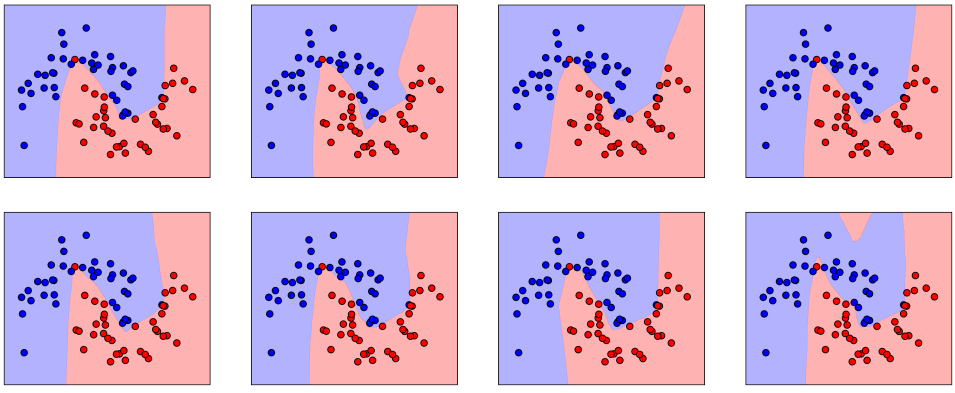
As you probably have realized by now, there are many ways to control the complexity of a neural network: the number of hidden layers, the number of units in each hidden layer, and the regularization (alpha). There are actually even more, which we won't go into here.

An important property of neural networks is that their weights are set randomly before learning is started, and this random initialization affects the model that is learned. That means that even when using exactly the same parameters, we can obtain very different models when using different random seeds.

If the networks are large, and their complexity is chosen properly, this should not affect accuracy too much, but it is worth keeping in mind (particularly for smaller networks).

Here are plots of several models, all learned with the same settings of the parameters:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(algorithm='l-bfgs', random_state=i,
                        hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=60, cmap=mglearn.cm2)
```



To get a better understanding of neural networks on real-world data, let's apply the `MLPClassifier` to the breast cancer dataset. We start with the default parameters:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

mlp = MLPClassifier()
mlp.fit(X_train, y_train)

print("accuracy on training set: %f" % mlp.score(X_train, y_train))
print("accuracy on test set: %f" % mlp.score(X_test, y_test))
```

```
accuracy on training set: 0.373239

accuracy on test set: 0.370629
```

As you can see, the result on both the training and the test set are devastatingly bad (even worse than random guessing!). As in the SVC example above, this is likely due to scaling of the data. Neural networks also expect all input features to vary in a similar way, and ideally should have a mean of zero, and a variance of one.

We [must] rescale our data so that it fulfills these requirements. Again, we will do this "by hand" here, but introduce the `StandardScaler` to do this automatically in Chapter 3 (Unsupervised Learning).

```
# compute the mean value per feature on the training set
mean_on_train = X_train.mean(axis=0)
# compute the standard deviation of each feature on the training set
std_on_train = X_train.std(axis=0)

# subtract the mean, scale by inverse standard deviation
# afterwards, mean=0 and std=1
X_train_scaled = (X_train - mean_on_train) / std_on_train
# use THE SAME transformation (using training mean and std) on the test set
X_test_scaled = (X_test - mean_on_train) / std_on_train

mlp = MLPClassifier(random_state=0)
```

```
mlp.fit(X_train_scaled, y_train)

print("accuracy on training set: %f" % mlp.score(X_train_scaled, y_train))
print("accuracy on test set: %f" % mlp.score(X_test_scaled, y_test))
```

```
/home/andy/checkout/scikit-learn/sklearn/neural_network/multilayer_perceptron.py:560: ConvergenceW

  % (), ConvergenceWarning)
```

The results are much better after scaling, and already quite competative. We got a warning from the model, though, that tells us that the maximum number of iterations has been reached. This is part of the `adam` algorithm for learning the model, and tells us that we should increase the number of iterations:

```
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("accuracy on training set: %f" % mlp.score(X_train_scaled, y_train))
print("accuracy on test set: %f" % mlp.score(X_test_scaled, y_test))
```

```
accuracy on training set: 0.995305

accuracy on test set: 0.965035
```

Increasing the number of iterations only increased the training set performance, but not the generalization performance. Still, the model is performing quite well. As there is some gap between the training

and the test performance, we might try to decrease the model complexity to get better generalization performance. Here, we choose to increase the alpha parameter (quite aggressively, from 0.0001 to 1), to add stronger regularization of the weights.

```
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("accuracy on training set: %f" % mlp.score(X_train_scaled, y_train))
print("accuracy on test set: %f" % mlp.score(X_test_scaled, y_test))
```

```
accuracy on training set: 0.988263

accuracy on test set: 0.972028
```

This leads to a performance on par with the best models so far. [Footnote: You might have noticed at this point that many of the well-performing models achieved exactly the same accuracy of 0.972. This means that all of the models make exactly the same number of mistakes, which is four. If you comparing the actual predictions, you can even see that they make exactly the same mistakes! This might be either a consequence of data being very small, or may be because these points are really different from the rest.]

While it is possible to analyze what a neural network learned, this is usually much trickier than analyzing a linear model or a tree-based model. One way to introspect

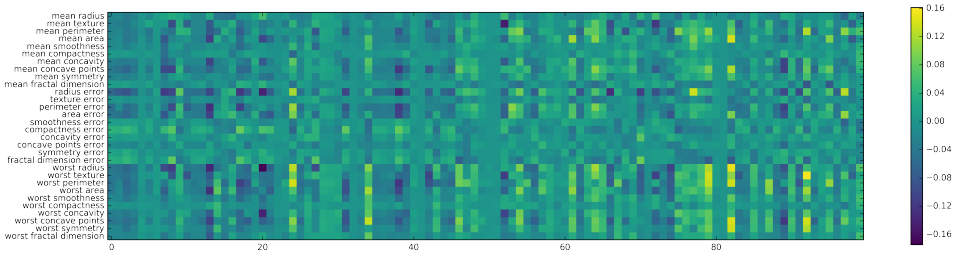
what was learned is to look at the weights in the model. You can see an example of this in the scikit-learn example gallery on the website. For the breast cancer dataset, this might be a bit hard to understand.

The plot below shows the weights that were learned connecting the input to the first hidden layer.

The rows in this plot correspond to the 30 input features, while the columns correspond to the 100 hidden units.

Light green represents large positive values, while dark blue represents negative values.

```
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.colorbar()
```



One possible inference we can make is that features that have very small weights for all of the hidden units are "less important" to the model. We can see that "mean smoothness" and "mean compactness" in addition to the features found between "smoothness error" and "fractal dimension error" have relatively low weights compared to other features. This could mean that these are less important features, or, possibly, that we didn't represent them in a way that the neural network could use.

We could also visualize the weights connecting the hidden layer to the output layer, but those are even harder to interpret.

While the `MLPClassifier` and `MLPRegressor` provide easy-to-use interfaces for the most common neural network architectures, they only capture a small subset of what is possible with neural networks. If you are interested in working with more flexible or larger models, we encourage you to look beyond scikit-learn into the fantastic deep learning libraries that are our there. For python users, the most well-established are keras, lasagna and tensor-flow. Keras and lasagna both build on the theano library.

These libraries provide a much more flexible interface to build neural networks, and track the rapid process in deep learning research. All of the popular deep learning libraries also allow the use of high-performance graphic processing units (GPUs), which scikit-learn does not support.

Using GPUs allows to accelerate computations by factors of 10x to 100x, and are essential for applying deep learning methods to large-scale datasets.

Strengths, weaknesses and parameters

Neural networks have re-emerged as state of the art models in many applications of machine learning. One of their main advantages is that they are able to capture information contained in large amounts of data and build incredibly complex models. Given enough computation time, data, and careful tuning of the parameters, neural networks often beat other machine learning algorithms (for classification and regression tasks).

This brings us to the downsides; neural networks, in particular the large and powerful ones, often take a long time to train. They also require careful preprocessing of the data, as we saw above. Similarly to SVMs, they work best with "homogeneous" data, where all the features have similar meanings. For data that has very different kinds of features, tree-based models might work better.

Tuning neural network parameters is also an art onto itself. In our experiments above, we barely scratched the surface of possible ways to adjust neural network models, and how to train them.

Estimating complexity in neural networks

The most important parameters are the number of layers and the number of hidden units per layer. You should start with one or two hidden layers, and possibly expand from there. The number of nodes per hidden layer is often around the number of the input features, but rarely higher than in the low to mid thousands.

A helpful measure when thinking about model complexity of a neural network is the number of weights or coefficients that are learned. If you have a binary classification dataset with 100 features, and you have 100 hidden units, then there are `100 * 100 = 10,000` weights between the input and the first hidden layer. There are also `100 * 1 = 100` weights between the hidden layer and the output layer, for a total of around `10,100` weights. If you add a second hidden layer with 100 hidden units, there will be another `100 * 100 = 10,000` weights from the first hidden layer to the second hidden layer, resulting in a total of `20,100` weights.

If instead, you use one layer with 1000 hidden units, you are learning `100 * 1000 = 100,000` weights from the input to the hidden layer, and `1000 x 1` weights from the hidden to the output layer, for a total of `101,000`.

If you add a second hidden layer, you add `1000 * 1000 = 1,000,000` weights, for a whopping `1,101,000`, which is 50 times larger than the model with two hidden layers of size 100.

A common way to adjust parameters in a neural network is to first create a network that is large enough to overfit, making sure that the task can actually be learned by the network. Once you know the training data can be learned, either shrink the network or increase alpha to add regularization, which will improve generalization performance.

During our experiments above, we focused mostly on the definition of the model: the number of layers and nodes per layer, the regularization, and the nonlinearity. These define the model we want to learn. There is also the question of *how* to learn the model, or the algorithm that is used for learning of the parameters, which is set using the `algorithm` parameter.

There are two easy-to-use choices for the `algorithm`. The default is `'adam'`, which works well in most situations but is quite sensitive to the scaling of the data (so it is important to always scale your data to zero mean and unit variance). The other one is `'l-bfgs'`, which is quite robust, but might take a long time on larger models or larger datasets.

There is also the more advanced `'sgd'` option, which is what many deep learning researchers use. The `'sgd'` option comes with many additional parameters that need to be tuned for best results. You can find all of these parameters and their definitions in the user-guide. When starting to work with MLPs, we recommend sticking to `adam` and `l-bfgs`.

Uncertainty estimates from classifiers

Another useful part of the scikit-learn interface that we haven't talked about yet is the ability of classifiers to provide uncertainty estimates of predictions.

Often, you are not only interested in which class a classifier predicts for a certain test point, but also how certain it is that this is the right class. In practice, different kinds of mistakes lead to very different outcomes in real world applications. Imagine a medical application testing for cancer. Making a false positive prediction might lead to a patient undergoing additional tests, while a false negative prediction might lead to a serious disease not being treated.

We will go into this topic in more detail in Chapter 6 (Model Selection).

There are two different functions in scikit-learn that can be used to obtain uncertainty estimates from classifiers, `decision_function` and `predict_proba`. Most (but not all) classifiers have at least one of them, and many classifiers have both. Let's look at what these two functions do on as synthetic two-dimensional dataset, when building a `GradientBoostingClassifier` classifier. `GradientBoostingClassifier` has both a `decision_function` method and a `predict_proba`.

```
# create and split a synthetic dataset
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_blobs, make_circles
# X, y = make_blobs(centers=2, random_state=59)
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)

# we rename the classes "blue" and "red" for illustration purposes:
y_named = np.array(["blue", "red"])[y]

# we can call train test split with arbitrary many arrays
# all will be split in a consistent manner
X_train, X_test, y_train_named, y_test_named, y_train, y_test = \
    train_test_split(X, y_named, y, random_state=0)

# build the gradient boosting model model
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train_named)
```

```
GradientBoostingClassifier(init=None, learning_rate=0.1, loss='deviance',

              max_depth=3, max_features=None, max_leaf_nodes=None,

              min_samples_leaf=1, min_samples_split=2,

              min_weight_fraction_leaf=0.0, n_estimators=100,

              presort='auto', random_state=0, subsample=1.0, verbose=0,

              warm_start=False)
```

The Decision Function

In the binary classification case, the return value of `decision_function` is of shape `(n_samples,)`, it returns one floating point number for each sample:

```
print(X_test.shape)
print(gbrt.decision_function(X_test).shape)
```

```
(25, 2)

(25,)
```

This value encodes how strongly the model believes a data point to belong to the "positive" class, in this case class 1.

Positive values indicate a preference for the positive class, negative values indicate preference for the "negative", that is the other class:

```
# show the first few entries of decision_function
gbrt.decision_function(X_test)[:6]
```

```
array([ 4.13592629, -1.68343075, -3.95106099, -3.6261613 ,  4.28986668,

        3.66166106])
```

We can recover the prediction by looking only at the sign of the decision function:

```
print(gbrt.decision_function(X_test) > 0)
print(gbrt.predict(X_test))
```

```
[ True False False False  True  True False  True  True  True False  True

  True False  True False False False  True  True  True  True  True False

 False]

['red' 'blue' 'blue' 'blue' 'red' 'red' 'blue' 'red' 'red' 'red' 'blue'

 'red' 'red' 'blue' 'red' 'blue' 'blue' 'blue' 'red' 'red' 'red' 'red'

 'red' 'blue' 'blue']
```

For binary classification, the "negative" class is always the first entry of the `classes_` attribute, and the "positive" class is the second entry of `classes_`. So if you want to fully recover the output of `predict`, you need to make use of the `classes_` attribute:

```
# make the boolean True/False into 0 and 1
greater_zero = (gbrt.decision_function(X_test) > 0).astype(int)
# use 0 and 1 as indices into classes_
pred = gbrt.classes_[greater_zero]
# pred is the same as the output of gbrt.predict
np.all(pred == gbrt.predict(X_test))
```

```
True
```

The range of `decision_function` can be arbitrary, and depends on the data and the model parameters:

```
decision_function = gbrt.decision_function(X_test)
np.min(decision_function), np.max(decision_function)
```

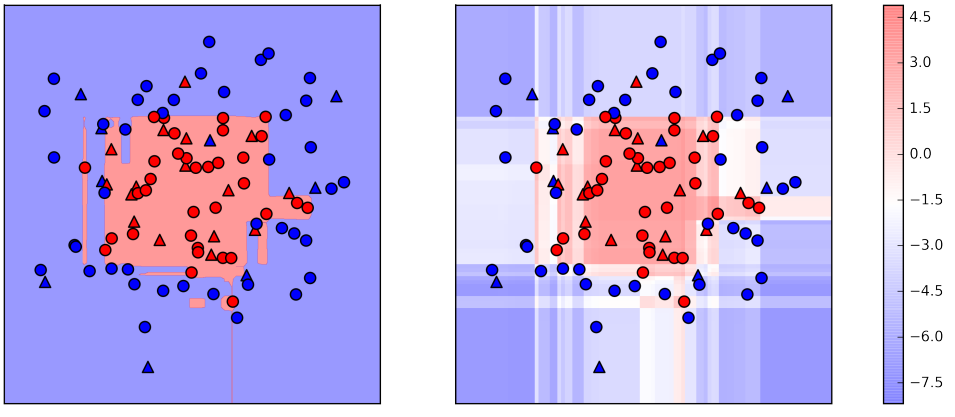
```
(-7.6909717730121798, 4.289866676868515)
```

This arbitrary scaling makes the output of `decision_function` often hard to interpret.

Below we plot the `decision_function` for all points in the 2d plane using a color coding, next to a visualization of the decision boundary, as we saw it in Chapter 2. We show training points as circles and test data as triangles.

Encoding not only the predicted outcome, but also how certain the classifier is provides additional information. However, in this visualization, it is hard to make out the boundary between the two classes.

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1], alpha=.4, cm='bwr')

for ax in axes:
    # plot training and test points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=mglearn.cm2, s=60, marker='^')
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=mglearn.cm2, s=60)
plt.colorbar(scores_image, ax=axes.tolist())
```



Predicting probabilities

The output of `predict_proba` however is a probability for each class, and is often more easily understood. It is always of shape `(n_samples, 2)` for binary classification:

```
gbrt.predict_proba(X_test).shape
```

```
(25, 2)
```

The first entry in each row is the estimated probability of the first class, the second entry is the estimated probability of the second class. Because it is a probability, the output of `predict_proba` is always between zero and 1, and the sum of the entries for both classes is always 1:

```
np.set_printoptions(suppress=True, precision=3)
# show the first few entries of predict_proba
gbrt.predict_proba(X_test[:6])
```

```
array([[ 0.016,  0.984],

       [ 0.843,  0.157],

       [ 0.981,  0.019],
```

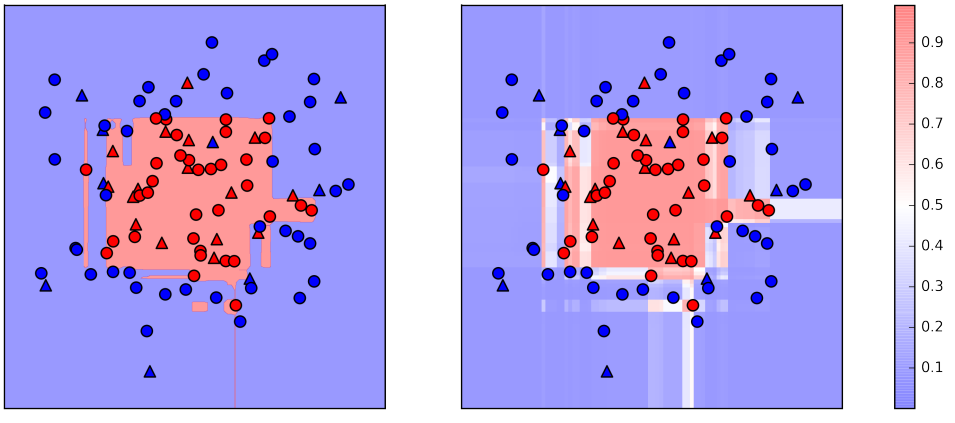
```
       [ 0.974,  0.026],

       [ 0.014,  0.986],

       [ 0.025,  0.975]])
```

Because the probabilities for the two classes sum to one, exactly one of the classes is above 50% certainty. That class is the one that is predicted.

You can see in the output above, that the classifier is relatively certain for most points. How well the uncertainty actually reflects uncertainty in the data depends on the model and parameters. A model that is more overfit tends to make more certain predictions, even if they might be wrong. A model with less complexity usually has more uncertainty in predictions. A model is called *calibrated* if the reported uncertainty actually matches how correct it is - in a calibrated model, a prediction made with 70% certainty would be correct 70% of the time.

Below we show again the decision boundary on the dataset, next to the class probabilities for the blue class:

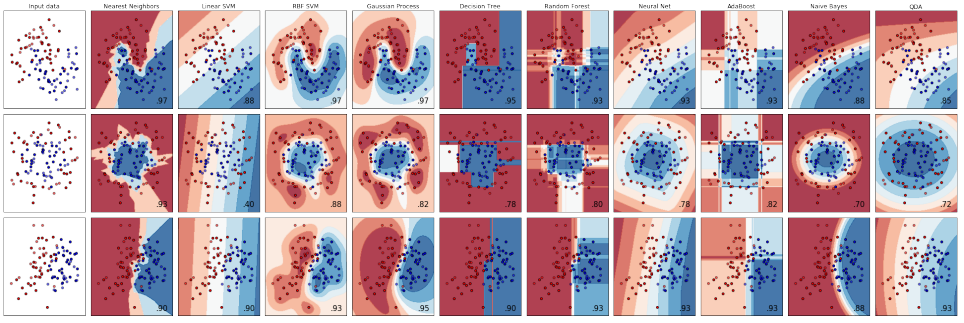
```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4,
                                fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1], alpha=.4,
                                            cm='bwr', function='predict_proba')

for ax in axes:
    # plot training and test points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=mglearn.cm2, s=60, marker='^')
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=mglearn.cm2, s=60)
plt.colorbar(scores_image, ax=axes.tolist())
```



The boundaries in this this plot are much more well-defined, and the small areas of uncertainty are clearly visible.

The scikit-learn website [Footnote: http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html] has a great comparison of many models, and how their uncertainty estimates look like.

We reproduced the figure below, and encourage you to go though the example there.



Uncertainty in multi-class classification

Above we only talked about uncertainty estimates in binary classification. But the `decision_function` and `predict_proba` methods also work in the multi-class setting.

Let's apply them on the `iris` dataset, which is a three-class classification dataset:

```
from sklearn.datasets import load_iris

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=42)

gbrt = GradientBoostingClassifier(learning_rate=0.01, random_state=0)
gbrt.fit(X_train, y_train)
```

```
GradientBoostingClassifier(init=None, learning_rate=0.01, loss='deviance',

              max_depth=3, max_features=None, max_leaf_nodes=None,

              min_samples_leaf=1, min_samples_split=2,

              min_weight_fraction_leaf=0.0, n_estimators=100,

              presort='auto', random_state=0, subsample=1.0, verbose=0,

              warm_start=False)
```

```
print(gbrt.decision_function(X_test).shape)
# plot the first few entries of the decision function
print(gbrt.decision_function(X_test)[:6, :])
```

```
(38, 3)
```

```
[[-0.529  1.466 -0.504]

 [ 1.512 -0.496 -0.503]

 [-0.524 -0.468  1.52 ]

 [-0.529  1.466 -0.504]

 [-0.531  1.282  0.215]

 [ 1.512 -0.496 -0.503]]
```

In the multi-class case, the `decision_function` has the shape `(n_samples, n_classes)`, and each column provides a "certainty score" for each class, where a large score means that a class is more likely, and a small score means the class is less likely. You can recover the prediction from these scores by finding the maximum entry for each data point:

```
print(np.argmax(gbrt.decision_function(X_test), axis=1))
print(gbrt.predict(X_test))
```

```
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1

 0]

[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1

 0]
```

The output of `predict_proba` has the same shape, `(n_samples, n_classes)`. Again, the probabilities for the possible classes for each data point sum to one:

```
# show the first few entries of predict_proba
print(gbrt.predict_proba(X_test)[:6])
# show that sums across rows are one
print("sums: %s" % gbrt.predict_proba(X_test)[:6].sum(axis=1))
```

```
[[ 0.107  0.784  0.109]

 [ 0.789  0.106  0.105]

 [ 0.102  0.108  0.789]

 [ 0.107  0.784  0.109]

 [ 0.108  0.663  0.228]

 [ 0.789  0.106  0.105]]

sums: [ 1.  1.  1.  1.  1.  1.]
```

We can again recover the predictions by computing the argmax of `predict_proba`:

```
print(np.argmax(gbrt.decision_function(X_test), axis=1))
print(gbrt.predict(X_test))
```

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

To summarize, `predict_proba` and `decision_function` always have shape `(n_samples, n_classes)` -- apart from the special case of `decision_function` in the binary case. In the binary case, `decision_function` only has one column, corresponding to the "positive" class `classes_[1]`. This is mostly for historical reasons.

You can recover the prediction when there are `n_classes` many columns by simply computing the argmax across columns.

Be careful, though, if your classes are strings, or you use integers, but they are not consecutive and starting from 0. If you want to compare results obtained with `predict` to results obtained via `decision_function` or `predict_proba` make sure to use the `classes_` attribute of the classifier to get the actual class names.

Summary and Outlook

We started this chapter with a discussion of model complexity, and discussed *generalization*, or learning a model that is able to perform well on new, unseen data. This led us to the concepts of underfitting, which describe a model that can not capture the variations present in the training data, and overfitting, which describe a model that focuses too much on the training data, and is not able to generalize to new data very well.

We then discussed a wide array of machine learning models for classification and regression, what their advantages and disadvantages are, and how to control model complexity for each of them.

We saw that for many of the algorithms, setting the right parameters is important for good performance. Some of the algorithms are also sensitive to how we represent the input data, in particular to how the features are scaled.

Therefore, blindly applying an algorithm to a dataset without understanding the assumptions the models makes and the meaning of the parameter settings will rarely lead to an accurate model.

This chapter contains a lot of information about the algorithms, and it is not necessary for you to remember all of these details for the following chapters. However, knowing the models described above, and knowing which to use in a specific situa-

tion, is important for successfully applying machine learning in practice. Here is a quick summary of when to use which model:

- Nearest neighbors: for small datasets, good as a baseline, easy to explain.
- Linear models: Go-to as a first algorithm to try, good for very large datasets, good for very high-dimensional data.
- Naive Bayes: Only for classification. Even faster than linear models, good for very large, high-dimensional data. Often less accurate than linear models.
- Decision trees: Very fast, don't need scaling of the data, can be visualized and easily explained.
- Random forests: Nearly always perform better than a single decision tree, very robust and powerful. Don't need scaling of data. Not good for very high-dimensional sparse data.
- Gradient Boosted Decision Trees: Often slightly more accurate than random forest. Slower to train but faster to predict than random forest, and smaller in memory. Need more parameter tuning than random forest.
- Support Vector Machines: Powerful for medium-sized datasets of features with similar meaning. Needs scaling of data, sensitive to parameters.
- Neural Networks: Can build very complex models, in particular for large datasets. Sensitive to scaling of the data, and to the choice of parameters. Large models need a long time to train.

When working with a new dataset, it is in general a good idea to start with a simple model, such as a linear model, naive Bayes or nearest neighbors and see how far you can get. After understanding more about the data, you can consider moving to an algorithm that can build more complex models, such as random forests, gradient boosting, SVMs or neural networks.

You should now be in a position where you have some idea how to apply, tune, and analyze the models we discussed above.

In this chapter, we focused on the binary classification case, as this is usually easiest to understand.

Most of the algorithms presented above have classification and regression variants, however, and all of the classification algorithms support both binary and multi-class classification.

Try to apply any of these algorithms to the build-in datasets in scikit-learn, like the `boston_housing` or `diabetes` datasets for regression, or the `digits` dataset for multi-class classification.

Playing around with the algorithms on different datasets will give you a better feel on how long they need to train, how easy it is to analyze the model, and how sensitive they are to the representation of the data.

While we analyzed the consequences of different parameter settings for the algorithms we investigated, building a model that actually generalizes well to new data in production is a bit trickier than that. We will see how to properly adjust parameters, and how to find good parameters automatically in Chapter 6 Model Selection.

Before we do this, we will dive in more detail into preprocessing and unsupervised learning in the next chapter.

CHAPTER 3

Unsupervised Learning and Preprocessing

The second family of machine learning algorithms that we will discuss is unsupervised learning.

Unsupervised learning subsumes all kinds of machine learning where there is no known output, no teacher to instruct the learning algorithm. In unsupervised learning, the learning algorithm is just shown the input data, and asked to extract knowledge from this data.

Types of unsupervised learning

We will look into two kinds of unsupervised learning in this chapter: transformations of the dataset, and clustering.

Unsupervised transformations of a dataset are algorithms that create a new representation of the data which might be easier for humans or other machine learning algorithms to understand.

A common application of unsupervised transformations is dimensionality reduction, which takes a high-dimensional representation of the data, consisting of many features, and finding a new way to represent this data that summarizes the essential characteristics about the data with fewer features. A common application for dimensionality reduction is reduction to two dimensions for visualization purposes.

Another application for unsupervised transformations is finding the parts or components that "make up" the data. An example of this is topic extraction on collections of text documents. Here, the task is to find the unknown *topics* that are talked about in each document, and to learn what topics appear in each document.

This can be useful for tracking the discussion of themes like elections, gun control or talk about pop-stars on social media.

Clustering algorithms on the other hand partition data into distinct groups of similar items.

Consider the example of uploading photos to a social media site. To allow you to organize your pictures, the site might want to group together pictures that show the same person. However, the site doesn't know which pictures show whom, and it doesn't know how many different people appear in your photo collection. A sensible approach would be to extract all faces, and divide them into groups of faces that look similar. Hopefully, these correspond to the same person, and can be grouped together for you.

Challenges in unsupervised learning

A major challenge in unsupervised learning is evaluating whether the algorithm learned something useful. Unsupervised learning algorithms are usually applied to data that does not contain any label information, so we don't know what the right output should be. Therefore it is very hard to say whether a model "did well". For example, the clustering algorithm could have grouped all face pictures that are shown in profile together, and all the face pictures that are face-forward together.

This would certainly be a possible way to divide a collection of face pictures, but not the one we were looking for. However, there is no way for us to "tell" the algorithm what we are looking for, and often the only way to evaluate the result of an unsupervised algorithm is to inspect it manually.

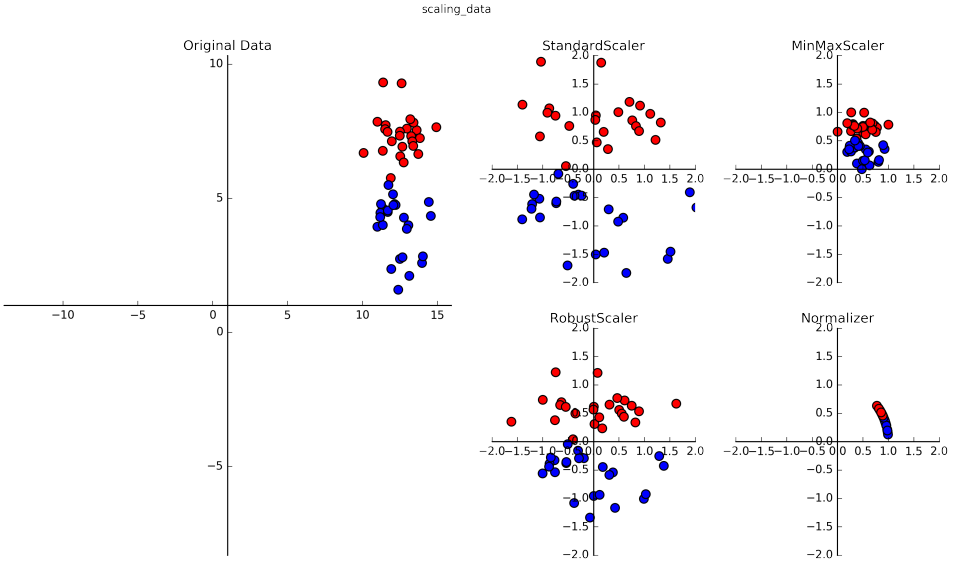
As a consequence, unsupervised algorithms are used often in an exploratory setting, when a data scientist wants to understand the data better, rather than as part of a larger automatic system. Another common application for unsupervised algorithms is as a preprocessing step for supervised algorithms. Learning a new representation of the data can sometimes improve the accuracy of supervised algorithms, or can lead to reduced memory and time consumption.

Before we start with "real" unsupervised algorithms, we will briefly discuss some simple preprocessing methods that often come in handy. Even though preprocessing and scaling are often used in tandem with supervised learning algorithms, scaling methods don't make use of the supervised information, making them unsupervised.

Preprocessing and Scaling

In the last chapter we saw that some algorithms, like neural networks and SVMs, are very sensitive to the scaling of the data. Therefore a common practice is to adjust the features so that the data representation is more suitable for these algorithms. Often, this is a simple per-feature rescaling and shift of the data. A simple example is shown in Figure scaling_data.

```
mglearn.plots.plot_scaling()
plt.suptitle("scaling_data");
```



Different kinds of preprocessing

The first plot shows a synthetic two-class classification dataset with two features. The first feature (the x-axis value) is between 10 and 15. The second feature (the y-axis value) is between around 1 and 9.

The following four plots show four different ways to transform the data that yield more standard ranges.

The `StandardScaler` in scikit-learn ensures that for each feature, the mean is zero, and the variance is one, bringing all features to the same magnitude. However, this scaling does not ensure any particular minimum and maximum values for the features.

The `RobustScaler` works similarly to the `StandardScaler` in that it ensures statistical properties for each feature that guarantee that they are on the same scale. However, the `RobustScaler` uses the median and quartiles [Footnote: the median of a set of numbers is the number x such that half of the numbers are smaller than x and half of the numbers are larger than x. The lower quartile is the number x such that 1/4th of the numbers are smaller than x, the upper quartile is so that 1/4th of the numbers is larger than x], instead of mean and variance. This makes the `RobustScaler` ignore data points that are very different from the rest (like measurement errors). These odd data points are also called *outliers*, and might often lead to trouble for other scaling techniques.

The MinMaxScaler on the other hand shifts the data such that all features are exactly between 0 and 1. For the two-dimensional dataset this means all of the data is contained within the rectangle created by the x axis between 0 and 1 and the y axis between zero and one.

Finally, the Normalizer does a very different kind of rescaling. It scales each data point such that the feature vector has a euclidean length of one. In other words, it projects a data point on the circle (or sphere in the case of higher dimensions) with a radius of 1. This means every data point is scaled by a different number (by the inverse of it's length).

This normalization is often used when only the direction (or angle) of the data matters, not the length of the feature vector.

Applying data transformations

After seeing what the different kind of transformations do, let's apply them using scikit-learn.

We will use the cancer dataset that we saw in chapter 2. Preprocessing methods like the scalers are usually applied before applying a supervised machine learning algorithm. As an example, say we want to apply the kernel SVM (SVC) to the cancer dataset, and use MinMaxScaler for preprocessing the data. We start by loading and splitting our dataset into a training set and a test set. We need a separate training and test set to evaluate the supervised model we will build after the preprocessing:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=1)
print(X_train.shape)
print(X_test.shape)
```

```
(426, 30)

(143, 30)
```

As a reminder, the data contains 150 data points, each represented by four measurements. We split the dataset into 112 samples for the training set and 38 samples for the test set.

As with the supervised models we built earlier, we first import the class implementing the preprocessing, and then instantiate it:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

We then fit the scaler using the `fit` method, applied to the training data. For the `MinMaxScaler`, the `fit` method computes the minimum and maximum value of each feature on the training set. In contrast to the classifiers and regressors of chapter 2, the scaler is only provided with the data `X_train` when `fit` is called, and `y_train` is not used:

```
scaler.fit(X_train)
```

```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

To apply the transformation that we just learned, that is, to actually *scale* the training data, we use the `transform` method of the scaler. The `transform` method is used in scikit-learn whenever a model returns a new representation of the data:

```
# don't print using scientific notation
np.set_printoptions(suppress=True, precision=2)
# transform data
X_train_scaled = scaler.transform(X_train)
# print data set properties before and after scaling
print("transformed shape: %s" % (X_train_scaled.shape,))
print("per-feature minimum before scaling:\n %s" % X_train.min(axis=0))
print("per-feature maximum before scaling:\n %s" % X_train.max(axis=0))
print("per-feature minimum after scaling:\n %s" % X_train_scaled.min(axis=0))
print("per-feature maximum after scaling:\n %s" % X_train_scaled.max(axis=0))
```

```
transformed shape: (426, 30)

per-feature minimum before scaling:

 [   6.98    9.71   43.79  143.5     0.05    0.02    0.      0.      0.11

    0.05    0.12    0.36    0.76    6.8     0.      0.      0.      0.

    0.01    0.      7.93   12.02   50.41  185.2     0.07    0.03    0.

    0.      0.16    0.06]

per-feature maximum before scaling:

 [   28.11    39.28   188.5   2501.       0.16     0.29     0.43     0.2

     0.3      0.1      2.87     4.88    21.98   542.2      0.03     0.14

     0.4      0.05     0.06     0.03    36.04    49.54   251.2   4254.

     0.22     0.94     1.17     0.29     0.58     0.15]

per-feature minimum after scaling:

 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.

  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

```
per-feature maximum after scaling:

 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.

   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

The transformed data has the same shape as the original data - the features are simply shifted and scaled.

You can see that all of the feature are now between zero and one, as desired.

To apply the SVM to the scaled data, we also need to transform the test set. This is done by again calling the transform method, this time on `X_test`:

```
# transform test data
X_test_scaled = scaler.transform(X_test)
# print test data properties after scaling
print("per-feature minimum after scaling: %s" % X_test_scaled.min(axis=0))
print("per-feature maximum after scaling: %s" % X_test_scaled.max(axis=0))
```

```
per-feature minimum after scaling: [ 0.03  0.02  0.03  0.01  0.14  0.04  0.    0.    0.15 -0.01 -0

  0.    0.    0.04  0.01  0.    0.   -0.03  0.01  0.03  0.06  0.02  0.01

  0.11  0.03  0.    0.   -0.   -0.  ]

per-feature maximum after scaling: [ 0.96  0.82  0.96  0.89  0.81  1.22  0.88  0.93  0.93  1.04  0

  0.44  0.28  0.49  0.74  0.77  0.63  1.34  0.39  0.9   0.79  0.85  0.74

  0.92  1.13  1.07  0.92  1.21  1.63]
```

Maybe somewhat surprisingly, you can see that for the test set, after scaling, the minimum and maximum are not zero and one. Some of the features are even outside the 0-1 range!

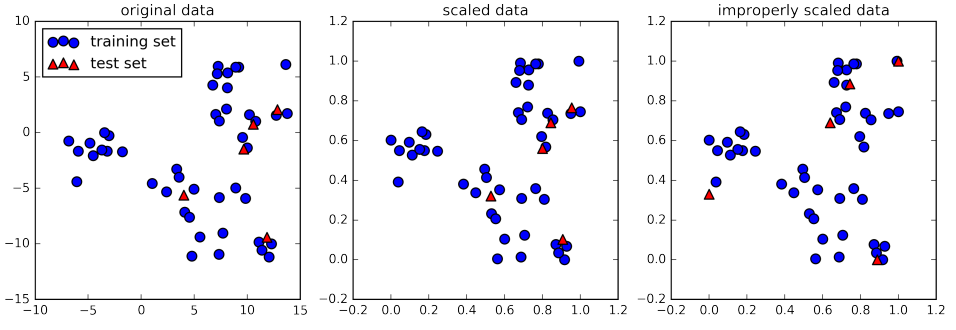
The explanation is that the `MinMaxScaler` (and all the other scalers) always applies exactly the same transformation to the training and the test set. So the `transform` method always subtracts the training set minimum, and divides by the training set range, which might be different than the minimum and range for the test set.

Scaling training and test data the same way

It is important that exactly the same transformation is applied to the training set and the test set for the supervised model to make sense on the test set. The following figure illustrates what would happen if we would use the minimum and range of the test set instead:

```
from sklearn.datasets import make_blobs
# make synthetic data
```

```
X, _ = make_blobs(n_samples=50, centers=5, random_state=4, cluster_std=2)
# split it into training and test set
X_train, X_test = train_test_split(X, random_state=5, test_size=.1)

# plot the training and test set
fig, axes = plt.subplots(1, 3, figsize=(13, 4))
axes[0].scatter(X_train[:, 0], X_train[:, 1],
                c='b', label="training set", s=60)
axes[0].scatter(X_test[:, 0], X_test[:, 1], marker='^',
                c='r', label="test set", s=60)
axes[0].legend(loc='upper left')
axes[0].set_title("original data")

# scale the data using MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# visualize the properly scaled data
axes[1].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c='b', label="training set", s=60)
axes[1].scatter(X_test_scaled[:, 0], X_test_scaled[:, 1], marker='^',
                c='r', label="test set", s=60)
axes[1].set_title("scaled data")

# rescale the test set separately, so that test set min is 0 and test set max is 1
# DO NOT DO THIS! For illustration purposes only
test_scaler = MinMaxScaler()
test_scaler.fit(X_test)
X_test_scaled_badly = test_scaler.transform(X_test)

# visualize wrongly scaled data
axes[2].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c='b', label="training set", s=60)
axes[2].scatter(X_test_scaled_badly[:, 0], X_test_scaled_badly[:, 1], marker='^',
                c='r', label="test set", s=60)
axes[2].set_title("improperly scaled data")
```



The first panel is an unscaled two-dimensional dataset, with the training set shown in blue and the test set shown in red. The second figure is the same data, but scaled using the `MinMaxScaler`. Here, we called `fit` on the training set, and then `transform` on the training and the test set. You can see that the dataset in the second panel looks identical to the first, only the ticks on the axes changed. Now all the features are between 0 and 1.

You can also see that the minimum and maximum feature values for the test data (the red points) are not 0 and 1.

The third panel shows what would happen if we scaled training and test set separately. In this case, the minimum and maximum feature values for both the training and the test set are 0 and 1. But now the dataset looks different. The test points moved incongruously to the training set, as they were scaled differently. We changed the arrangement of the data in an arbitrary way. Clearly this is not what we want to do.

Another way to reason about this is the following: Imagine your test set was a single point. There is no way to scale a single point correctly, to fulfill the minimum and maximum requirements of the `MinMaxScaler`. But the size of your test set should not change your processing.

The effect of preprocessing on supervised learning

Now let's go back to the cancer dataset and see what the effect of using the `MinMaxScaler` is on learning the `SVC` (this is a different way of doing the same scaling we did in chapter 2).

First, let's fit the `SVC` on the original data again for comparison:

```
from sklearn.svm import SVC

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=0)

svm = SVC(C=100)
svm.fit(X_train, y_train)
print(svm.score(X_test, y_test))
```

```
0.629370629371
```

Now, let's scale the data using `MinMaxScaler` before fitting the `SVC`:

```
# preprocessing using 0-1 scaling
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# learning an SVM on the scaled training data
```

```
svm.fit(X_train_scaled, y_train)
# scoring on the scaled test set

svm.score(X_test_scaled, y_test)
```

```
0.965034965034965
```

As we saw before, the effect of scaling the data is quite significant. Even though scaling the data doesn't involve any complicated math, it is good practice to use the scaling mechanisms provided by scikit-learn, instead of reimplementing them yourself, as making mistakes even in these simple computations is easy.

You can also easily replace one preprocessing algorithm by another by changing the class you use, as all of the preprocessing classes have the same interface, consisting of the `fit` and `transform` methods:

```
# preprocessing using zero mean and unit variance scaling
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# learning an SVM on the scaled training data
svm.fit(X_train_scaled, y_train)
# scoring on the scaled test set
svm.score(X_test_scaled, y_test)
```

```
0.95804195804195802
```

Now that we've seen how simple data transformations for preprocessing work, let's move on to more interesting transformations using unsupervised learning.

Dimensionality Reduction, Feature Extraction and Manifold Learning

As we discussed above, transforming data using unsupervised learning can have many motivations. The most common motivations are visualization, compressing the data, and finding a representation that is more informative for further processing.

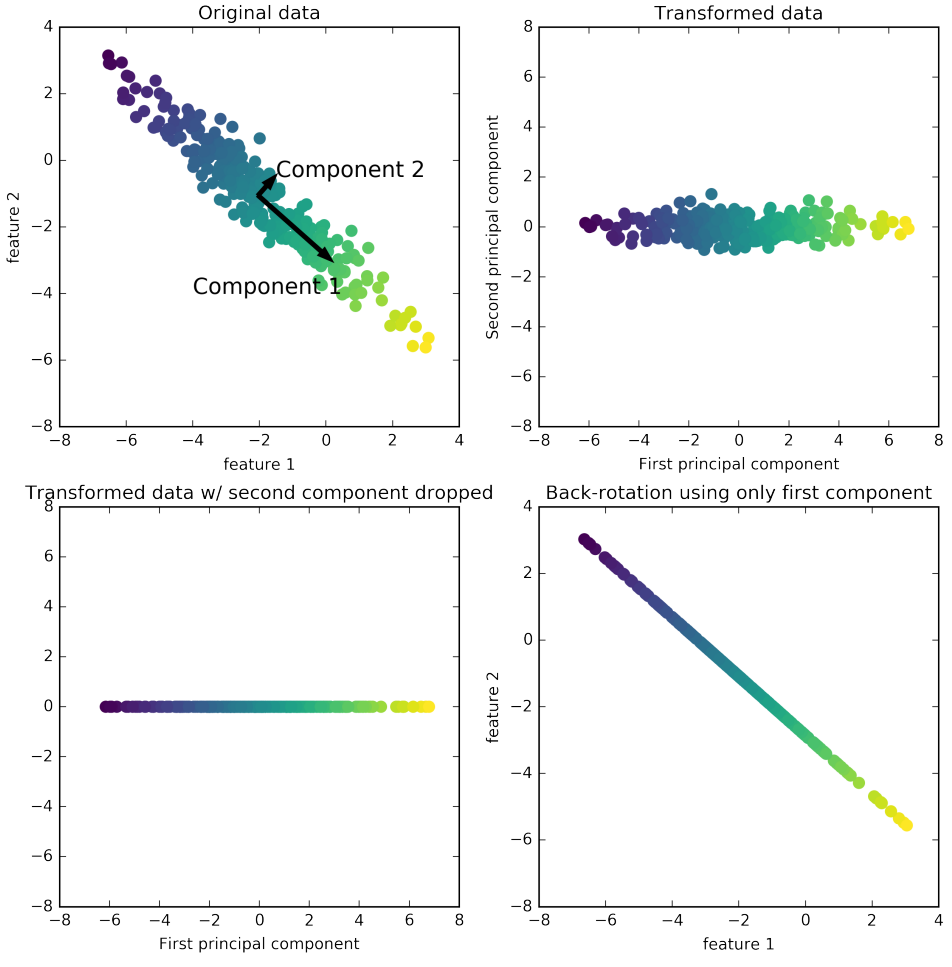
One of the simplest and most widely used algorithms for all of these is Principal Component Analysis.

Principal Component Analysis (PCA)

Principal component analysis (PCA) is a method that rotates the dataset in a way such that the rotated features are statistically uncorrelated. This rotation is often followed by selecting only a subset of the new features, according to how important they are for explaining the data.

```
mglearn.plots.plot_pca_illustration()
plt.suptitle("pca_illustration");
```

pca_illustration



The plot above shows a simple example on a synthetic two-dimensional dataset. The first plot shows the original data points, colored to distinguish the points. The algorithm proceeds by first finding the direction of maximum variance, labeled as "Component 1". This is the direction in the data that contains most of the information, or in other words, the direction along which the features are most correlated with each other.

Then, the algorithm finds the direction that contains the most information while being orthogonal (is at a right angle) to the first direction. In two dimensions, there is

only one possible orientation that is at a right angle, but in higher dimensional spaces there would be (infinitely) many orthogonal directions.

Although the two components are drawn as arrows, it doesn't really matter where the head and the tail is; we could have drawn the first component from the center up to the top left instead of to the bottom right.The directions found using this process are called *principal components*, as they are the main directions of variance in the data. In general, there are as many principal components as original features.

The second plot shows the same data, but now rotated so that the first principal component aligns with the x axis, and the second principal component aligns with the y axis. Before the rotation, the mean was subtracted from the data, so that the transformed data is centered around zero. In the rotated representation found by PCA, the two axes are uncorrelated, meaning that the correlation matrix of the data in this representation is zero except for the diagonal.

We can use PCA for dimensionality reduction by retaining only some of the principal components. In this example, we might keep only the first principal component, as shown in the third panel in Figure X.

This reduced the data from a two-dimensional dataset to a one-dimensional dataset. But instead of keeping only one of the original features, we found the most interesting direction (top left to bottom right in the first panel) and kept this direction, the first principal component.

Finally, we can undo the rotation, and add the mean back to the data. This will result in the data shown in the last panel. These points are in the original feature space, but we kept only the information contained in the first principal component. This transformation is sometimes used to remove noise effects from the data, or visualize what part of the information is kept in the PCA.

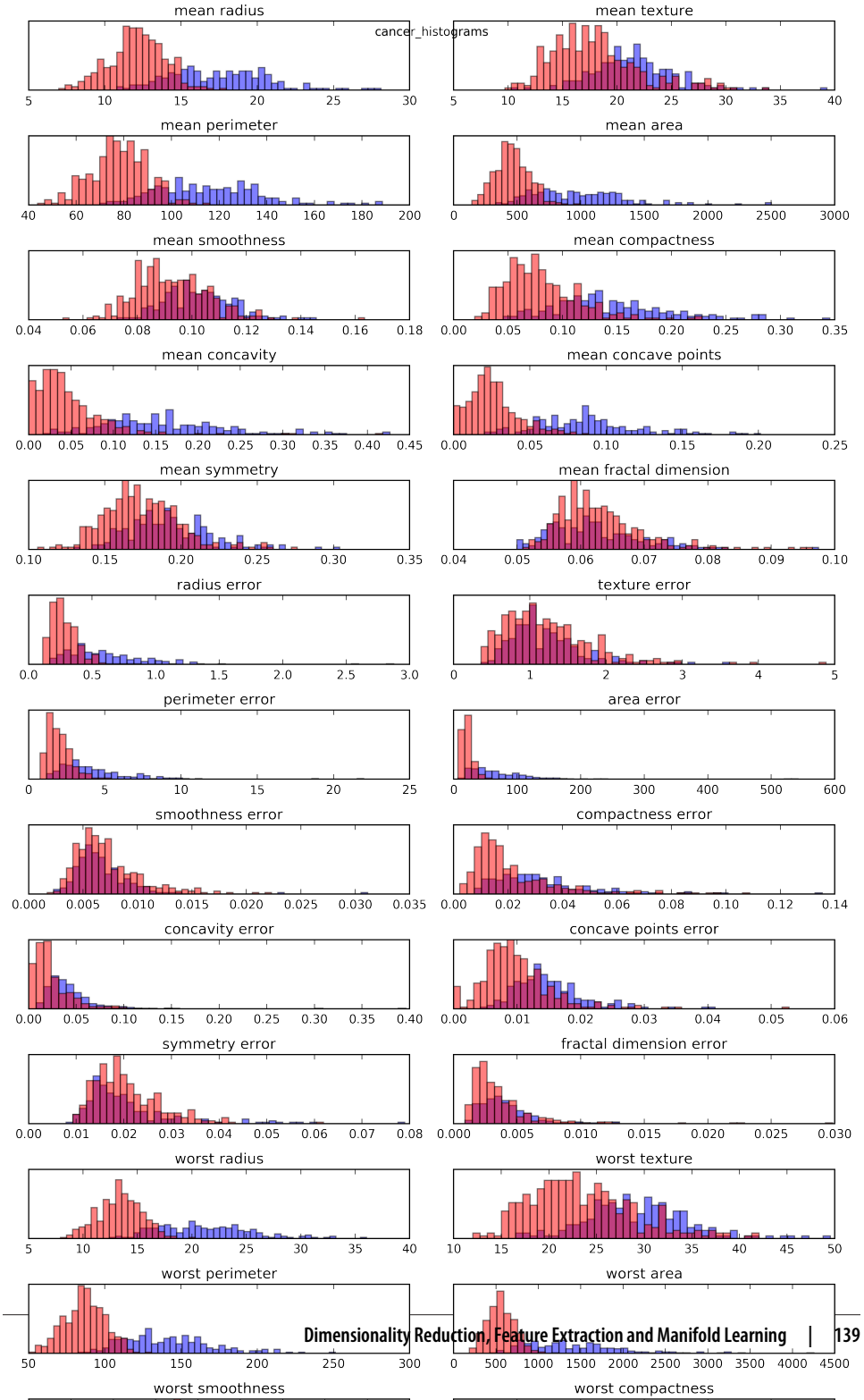
Applying PCA to the cancer dataset for visualization

One of the most common applications of PCA is visualizing high-dimensional datasets. As we already saw in Chapter 1, it is hard to create scatter plots of data that has more than two features. For the iris dataset, we could create a pair plot (Figure iris_pairplot in Chapter 1), which gave us us a partial picture of the data by showing us all combinations of two features. If we want to look at the breast cancer dataset, even using a pair-plot is tricky. The breast cancer dataset has 30 features, which would result in 30 * 14 = 420 scatter plots! You'd never be able to look at all these plots in detail, let alone try to understand them.

We could go for an even simpler visualization, showing histograms of each of the features for the two classes, benign and malignant cancer:

```
fig, axes = plt.subplots(15, 2, figsize=(10, 20))
malignant = cancer.data[cancer.target == 0]
```

```
benign = cancer.data[cancer.target == 1]

ax = axes.ravel()

for i in range(30):
    _, bins = np.histogram(cancer.data[:, i], bins=50)
    ax[i].hist(malignant[:, i], bins=bins, color='b', alpha=.5)
    ax[i].hist(benign[:, i], bins=bins, color='r', alpha=.5)
    ax[i].set_title(cancer.feature_names[i])
    ax[i].set_yticks(())
fig.tight_layout()
plt.suptitle("cancer_histograms")
```



Here we create a histogram for each of the features, counting how often a data point appears with a feature in a certain range (called a bin).

Each plot overlays two histograms, one for all of the points of the benign class (blue) and one for all the points in the malignant class (red). This gives us some idea of how each feature is distributed across the two classes, and allows us to venture a guess as to which features are better at distinguishing malignant and benign samples. For example, the feature "smoothness error" seems quite uninformative, because the two histograms mostly overlap, while the feature "worst concave points" seems quite informative, because the histograms are quite disjoint.

However, this plot doesn't show us anything about the, which indicate variables that are varying together . We can find the first two principal components, and visualize the data in this new, two-dimensional space, with a single scatter-plot.

Before we apply PCA, we scale our data so that each feature has unit variance using `StandardScaler`:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()

scaler = StandardScaler()
scaler.fit(cancer.data)
X_scaled = scaler.transform(cancer.data)
```

Learning the PCA transformation and applying it is as simple as applying a preprocessing transformation. We instantiate the PCA object, find the principal components by calling the `fit` method, and then apply the rotation and dimensionality reduction by calling `transform`.

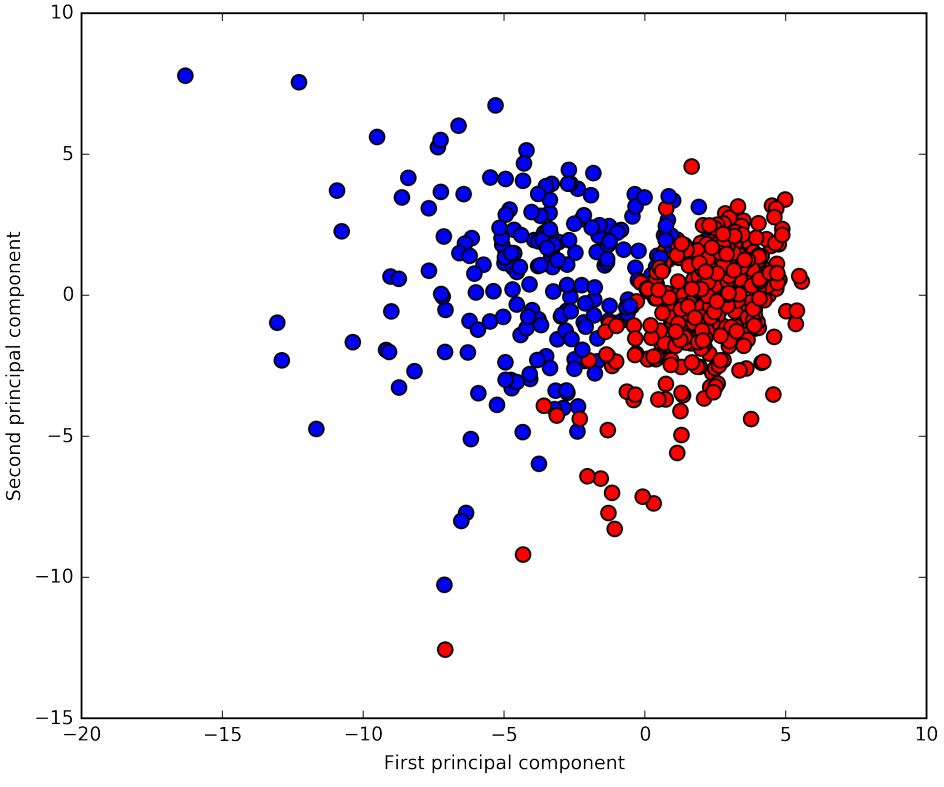
By default, `PCA` only rotates (and shifts) the data, but keeps all principal components. To reduce the dimensionality of the data, we need to specify how many components we want to keep when creating the `PCA` object:

```
from sklearn.decomposition import PCA
# keep the first two principal components of the data
pca = PCA(n_components=2)
# fit PCA model to beast cancer data
pca.fit(X_scaled)

# transform data onto the first two principal components
X_pca = pca.transform(X_scaled)
print("Original shape: %s" % str(X_scaled.shape))
print("Reduced shape: %s" % str(X_pca.shape))
```

```
Original shape: (569, 30)

Reduced shape: (569, 2)
```

We can now plot the first two principal components:

```
# plot fist vs second principal component, color by class
plt.figure(figsize=(8, 8))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=cancer.target, cmap=mglearn.tools.cm, s=60)
plt.gca().set_aspect("equal")
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```



It is important to note is that PCA is an unsupervised method, and does not use any class information when finding the rotation. It simply looks at the correlations in the data. For the scatter plot above, we plotted the first principal component against the second principal component, and then used the class information to color the points.

You can see that the two classes separate quite well in this two-dimensional space. This can lead us to believe that even a linear classifier (that would learn a line in this space) could do a reasonably good job at distinguishing the two classes. We can also see that the malignant (red) points are more spread-out than the benign (blue) points, something that we could already see a bit from the histograms in Figure cancer_histograms.

A downside of PCA is that the two axes in the plot above are often not very easy to interpret. The principal components correspond to directions in the original data, so they are combinations of the original features. However, these combinations are usually very complex, as we'll see below.

The principal components themselves are stored in the `components_` attribute of the PCA during fitting:

```
pca.components_.shape
```

```
(2, 30)
```

Each row in `components_` corresponds to one principal component, sorted by their importance (the first principal component comes first, etc). The columns correspond to the original features, in this example "mean radius", "mean texture" and so on.

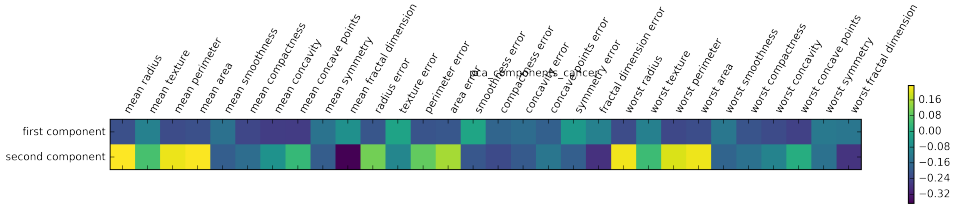
Let's have a look at the content of `components_`:

```
print(pca.components_)
```

```
[[-0.22 -0.1  -0.23 -0.22 -0.14 -0.24 -0.26 -0.26 -0.14 -0.06 -0.21 -0.02
  -0.21 -0.2  -0.01 -0.17 -0.15 -0.18 -0.04 -0.1  -0.23 -0.1  -0.24 -0.22
  -0.13 -0.21 -0.23 -0.25 -0.12 -0.13]
 [ 0.23  0.06  0.22  0.23 -0.19 -0.15 -0.06  0.03 -0.19 -0.37  0.11 -0.09
   0.09  0.15 -0.2  -0.23 -0.2  -0.13 -0.18 -0.28  0.22  0.05  0.2   0.22
  -0.17 -0.14 -0.1   0.01 -0.14 -0.28]]
```

We can also visualize the coefficients using a heatmap, which might be easier to understand:

```
plt.matshow(pca.components_, cmap='viridis')
plt.yticks([0, 1], ["first component", "second component"])
plt.colorbar()
plt.xticks(range(len(cancer.feature_names)),
           cancer.feature_names, rotation=60, ha='left');
plt.suptitle("pca_components_cancer")
```



You can see that in the first component, all feature have the same sign (it's negative, but as we mentioned above, it doesn't matter in which direction you point the arrow).

That means that there is a general correlation between all features. As one measurement is high, the others are likely to be high as well.

The second component has mixed signs, and both of the components involve all of the 30 features. This mixing of all features is what makes explaining the axes in Figure pca_components_cancer above so tricky.

Eigenfaces for feature extraction

Another application of PCA that we mentioned above is feature extraction. The idea behind feature extraction is that it is possible to find a representation of your data that is better suited to analysis than the raw representation you were given. A great example of an application when feature extraction if helpful is with images. Images are usually stored as red, green and blue intensities for each pixel. But images are made up of many pixels, and only together are they meaningful; objects in images are usually made up of thousands of pixels.

We will give a very simple application of feature extraction on images using PCA, using face images from the "labeled faces in the wild" dataset. This dataset contains face images of celebrities downloaded from the internet, and it includes faces of politicians, singers, actors and athletes from the early 2000s. We use gray-scale versions of these images, and scale them down for faster processing. You can see some of the images below:

```
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
image_shape = people.images[0].shape

fix, axes = plt.subplots(2, 5, figsize=(15, 8), subplot_kw={'xticks': (), 'yticks': ()})
for target, image, ax in zip(people.target, people.images, axes.ravel()):
    ax.imshow(image)
    ax.set_title(people.target_names[target])
plt.suptitle("some_faces")
```

There are 3023 images, each 87 x 65 pixels large, belonging to 62 different people:

```
print(people.images.shape)
print(len(people.target_names))
```

```
(3023, 87, 65)

62
```

The dataset is a bit skewed, however, containing a lot of images of George W. Bush and Colin Powell, as you can see here:

```
# count how often each target appears
counts = np.bincount(people.target)
# print counts next to target names:
for i, (count, name) in enumerate(zip(counts, people.target_names)):
    print("{0:25} {1:3}".format(name, count), end='   ')
    if (i + 1) % 3 == 0:
        print()
```

```
Alejandro Toledo           39   Alvaro Uribe              35   Amelie Mauresmo           21
Andre Agassi               36   Angelina Jolie            20   Ariel Sharon              77
Arnold Schwarzenegger      42   Atal Bihari Vajpayee      24   Bill Clinton              29
Carlos Menem               21   Colin Powell             236   David Beckham             31
Donald Rumsfeld           121   George Robertson          22   George W Bush            530
Gerhard Schroeder         109   Gloria Macapagal Arroyo   44   Gray Davis                26
```

```
Guillermo Coria           30   Hamid Karzai              22   Hans Blix                 39
Hugo Chavez               71   Igor Ivanov               20   Jack Straw                28
Jacques Chirac            52   Jean Chretien             55   Jennifer Aniston          21
Jennifer Capriati         42   Jennifer Lopez            21   Jeremy Greenstock         24
Jiang Zemin               20   John Ashcroft             53   John Negroponte           31
Jose Maria Aznar          23   Juan Carlos Ferrero       28   Junichiro Koizumi         60
Kofi Annan                32   Laura Bush                41   Lindsay Davenport         22
Lleyton Hewitt            41   Luiz Inacio Lula da Silva 48   Mahmoud Abbas             29
Megawati Sukarnoputri     33   Michael Bloomberg         20   Naomi Watts               22
Nestor Kirchner           37   Paul Bremer               20   Pete Sampras              22
Recep Tayyip Erdogan      30   Ricardo Lagos             27   Roh Moo-hyun              32
Rudolph Giuliani          26   Saddam Hussein            23   Serena Williams           52
Silvio Berlusconi         33   Tiger Woods               23   Tom Daschle               25
Tom Ridge                 33   Tony Blair               144   Vicente Fox               32
Vladimir Putin            49   Winona Ryder              24
```

To make the data less skewed, we will only take up to 50 images of each person. Otherwise the feature extraction would be overwhelmed by the likelihood of George W Bush.

```
mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:50]] = 1

X_people = people.data[mask]
y_people = people.target[mask]

# scale the grey-scale values to be between 0 and 1
# instead of 0 and 255 for better numeric stability:
X_people = X_people / 255.
```

A common task in face recognition is to ask if a previously unseen face belongs to a known person from a database. This has applications in photo collection, social media and security. One way to solve this problem would be to build a classifier where each person is a separate class. However, there are usually many different people in face databases, and very few images of the same person (i.e. very few training

examples per class). That makes it hard to train most classifiers. Additionally, you often want to easily add new people, without retraining a large model.

A simple solution is to use a one-nearest-neighbor classifier which looks for the most similar face image to the face you are classifying. A one-nearest-neighbor could in principle work with only a single training example per class. Let's see how well `KNeighborsClassifier` does here:

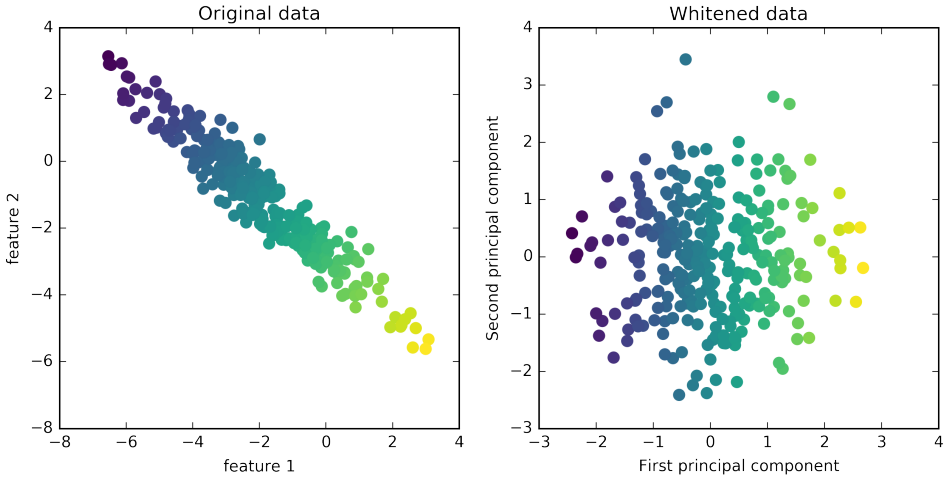
```
from sklearn.neighbors import KNeighborsClassifier
# split the data in training and test set
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
# build a KNeighborsClassifier with using one neighbor:
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
knn.score(X_test, y_test)
```

```
0.26615969581749049
```

We obtain an accuracy of 26.6%, which is not actually that bad for a 62 class classification problem (random guessing would give you around 1/62 = 1.5% accuracy), but is also not great. We only correctly identify a person a every fourth time.

This is where PCA comes in. Computing distances in the original pixel space is quite a bad way to measure similarity between faces [add a sentence saying why]. We hope that using distances along principal components can improve our accuracy. Here we enable the *whitening* option of PCA, which rescales the principal components to have the same scale. This is the same as using `StandardScaler` after the transformation. Reusing the data from Figure pca_illustration again, whitening corresponds to not only rotating the data, but also rescaling it so that the center panel is a circle instead of an ellipse:

```
mglearn.plots.plot_pca_whitening()
```



We fit the PCA object to the training data and extract the first 100 principal components. Then we transform the training and test data:

```
pca = PCA(n_components=100, whiten=True).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

print(X_train_pca.shape)
```

```
(1537, 100)
```

The new data has 100 features, the first 100 principal components. Now, we can use the new representation to classify our images using one-nearest-neighbors:

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_pca, y_train)
knn.score(X_test_pca, y_test)
```

```
0.36882129277566539
```

Our accuracy improved quite significantly, from 26.6% to 36.8%, confirming our intuition that the principal components might provide a better representation of the data.

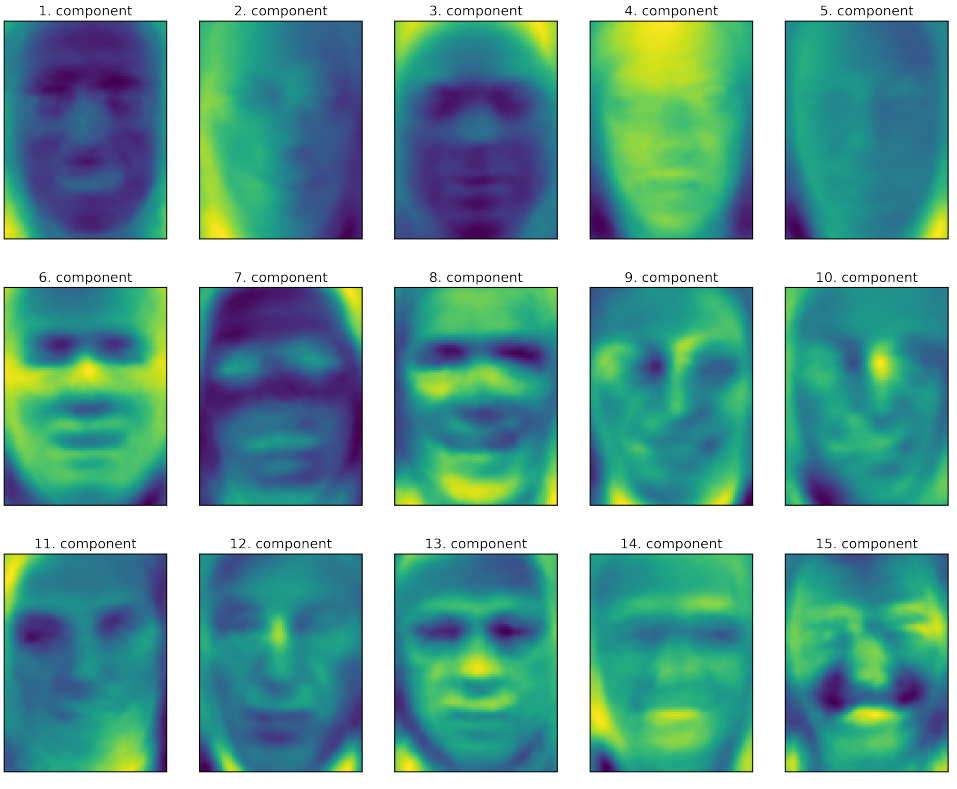
For image data, we can also easily visualize the principal components that are found. Remember that components correspond to directions in the input space. The input space here is 50x37 gray-scale images, and so directions within this space are also 50x37 gray-scale images. Let's look at the first couple of principal components:

```
pca.components_.shape
```

```
(100, 5655)
```

```
fix, axes = plt.subplots(3, 5, figsize=(15, 12),
                         subplot_kw={'xticks': (), 'yticks': ()})
```

```
fig.suptitle("pca_face_components")
for i, (component, ax) in enumerate(zip(pca.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape),
              cmap='viridis')
    ax.set_title("%d. component" % (i + 1))
```



While we certainly can not understand all aspects of these components, we can guess which aspects of the face images some of the components are capturing. The first component seems to mostly encode the contrast between the face and the background, the second component encodes differences in lighting between the right and the left half of the face, and so on. While this representation is slightly more semantic than the raw pixel values, it is still quite far from how a human might perceive a face. As the PCA is based on pixels, the alignment of the face (the position of eyes, chin and nose), as well as the lighting, both have a strong influence on how similar to images are in their pixel representation. Alignment and lighting are probably not what a human would perceive first. When asking a person to rate similarity of faces, they are more likely to use attributes like age, gender, facial expression and hair style, which are attributes that are hard to infer from the pixel intensities.

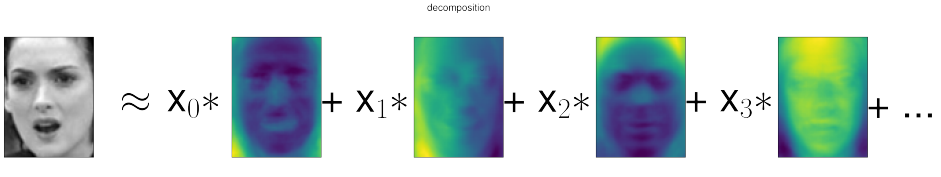
It's important to keep in mind that algorithms often interpret data, in particular data that humans are used to understand, like images, quite differently from how a human would.

Let's come back to the specific case of PCA, though.

Above we introduced the PCA transformation as rotating the data, and then dropping the components with low variance.

Another useful interpretation is that we try to find some numbers (the new feature values after the PCA rotation), so that we can express the test points as a weighted sum of the principal components:

```
from matplotlib.offsetbox import OffsetImage, AnnotationBbox

image_shape = people.images[0].shape
plt.figure(figsize=(20, 3))
ax = plt.gca()

imagebox = OffsetImage(people.images[0], zoom=7, cmap="gray")
ab = AnnotationBbox(imagebox, (.05, 0.4), pad=0.0, xycoords='data')
ax.add_artist(ab)

for i in range(4):
    imagebox = OffsetImage(pca.components_[i].reshape(image_shape), zoom=7, cmap="viridis")

    ab = AnnotationBbox(imagebox, (.3 + .2 * i, 0.4),
                        pad=0.0,
                        xycoords='data'
                        )
    ax.add_artist(ab)
    if i == 0:
        plt.text(.18, .25, 'x_%d *' % i, fontdict={'fontsize': 50})
    else:
        plt.text(.15 + .2 * i, .25, '+ x_%d *' % i, fontdict={'fontsize': 50})

plt.text(.95, .25, '+ ...', fontdict={'fontsize': 50})

plt.rc('text', usetex=True)
plt.text(.13, .3, r'\approx', fontdict={'fontsize': 50})
plt.axis("off")
plt.title("decomposition")
```




```
plt.rc('text', usetex=False) # THIS SHOULD NOT SHOW IN THE BOOK! it's needed for the figure above
```

Here, x_0, x_1 and so on are the coefficients of the principal components for this data point; in other words, they are the representation [of what? of the image? this is not clear] in the rotated space.

Another way we can try to understand what a PCA model is doing is by looking at the reconstructions of the original data using only some components. In Figure pca_illustration, after dropping the second component and arriving at the third panel , we undid the rotation and added the mean back to obtain new points in the original space with the second component removed, as shown in the last panel.

We can do a similar transformation for the faces by reducing the data to only some principal components and then rotating back into the original space. This return to the original feature space can be done using the `inverse_transform` method. Here we visualize the reconstruction of some faces using 10, 50, 100, 500 or 2000 components:

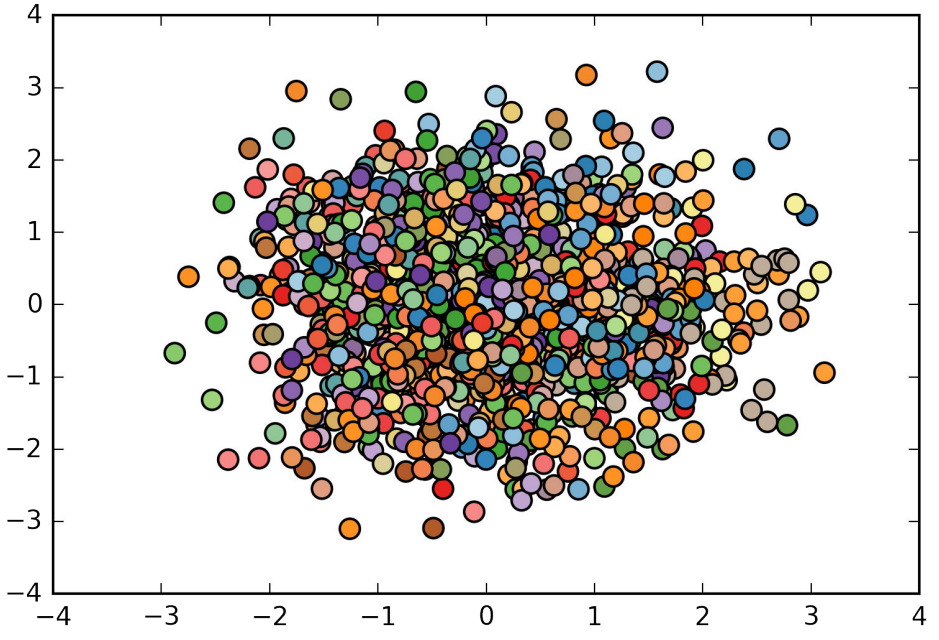
```
mglearn.plots.plot_pca_faces(X_train, X_test, image_shape)
plt.suptitle("pca_reconstructions");
```

```
/home/andy/checkout/scikit-learn/sklearn/externals/joblib/logger.py:77: DeprecationWarning: The 'w

  logging.warn("[%s]: %s" % (self, msg))

________________________________________________________________________________

[Memory] Calling mglearn.plot_pca.pca_faces...

pca_faces(array([[ 0.036601, ...,  0.742484],

       ...,

       [ 0.105882, ...,  0.393464]], dtype=float32),

array([[ 0.162091, ...,  0.677124],

       ...,

       [ 0.109804, ...,  0.07451 ]], dtype=float32))

_____________________________________________________pca_faces - 12.9s, 0.2min
```

You can see that with using only the first 10 principal components, only the essence of the picture, like the face orientation and lighting, is captured. Using more and more principal components, more and more details in the image are preserved. This corresponds to extending the sum in Figure decomposition to include more and more terms. Using as many components as there are pixels would mean that we would not discard any information after the rotation, and we would reconstruct the image perfectly.

We can also try to use PCA to visualize all the faces in the dataset in a scatter plot using the first two principal components, with classes given by who is shown in the image, similarly to what we did for the cancer dataset:

```
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1], c=y_train, cmap='Paired', s=60)
```



As you can see, when using just the first two principal components, the whole data is just a big blob, with no separation of classes visible. This is not very surprising, given that even with 10 components, as shown in Figure faces_pca_reconstruction above, PCA only captures very rough characteristics of the faces.

Non-Negative Matrix Factorization (NMF)

Non-negative matrix factorization is another unsupervised learning algorithm that aims to extract useful features. It works similarly to PCA and can also be used for dimensionality reduction. As in PCA we are trying to write each data point as a weighted sum of some components as illustrated in Figure decomposition. In PCA, we wanted components that are orthogonal, and that explain as much variance of the data as possible. In NMF, we want the components and the coefficients to be non-negative; that is, we want both the components and the coefficients to be greater or equal then zero.

Consequently, this method can only applied to data where each feature is non-negative, as a non-negative sum of non-negative components can not become negative. The process of decomposing data into a non-negative weighted sum is particularly helpful for data that is created as the addition of several independent sources, such as an audio track of multiple speakers, or music with many instruments. In these situations, NMF can identify the original components that make up the combined data. Overall, NMF leads to more interpretable components than PCA, as neg-

ative components and coefficients can lead to hard-interpret cancellation effects. The eigenfaces in Figure pca_face_components for example contain both positive and negative parts, and as we mentioned in the description of PCA, the sign is actually arbitrary.

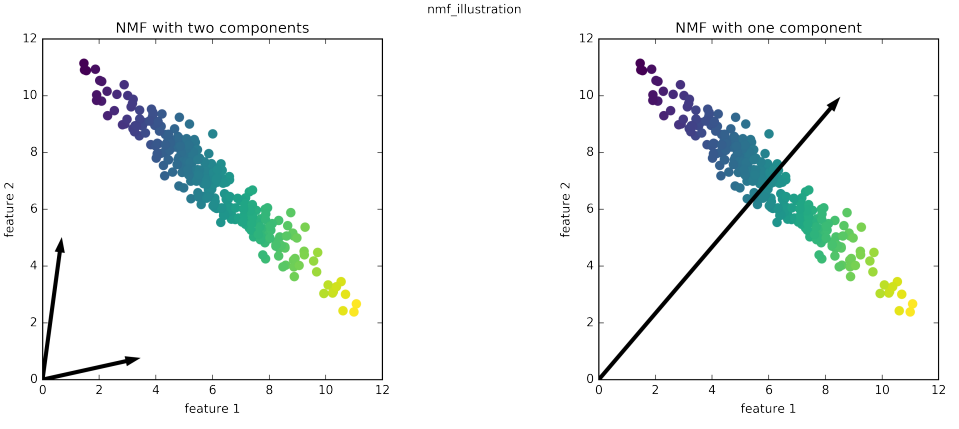
Before we apply NMF to the face dataset, let's briefly revisit the synthetic data.

Applying NMF to synthetic data

In contrast to PCA, we need to ensure that our data is positive for NMF to be able to operate on the data.

This means where the data lies relative to the origin (0, 0) actually matters for NMF. Therefore, you can think of the non-negative components that are extracted as directions from (0, 0) towards the data.

```
mglearn.plots.plot_nmf_illustration()
plt.suptitle("nmf_illustration")
```



The figure above shows the results of NMF on the two-dimensional toy data. For NMF with two components, as shown on the left, it is clear that all points in the data can be written as a positive combination of the two components. If there are enough components to perfectly reconstruct the data (as many components are there are features), the algorithm will choose directions that point towards the extremes of the data.

If we only use a single component, NMF creates a component that points towards the mean, as pointing there best explains the data. You see that in contrast to PCA, reducing the number of components not only removes directions, it actually changes all directions! Components in NMF are also not ordered in any specific way, so there is no "first non-negative component": all components play an equal part.

NMF uses a random initialization, which might lead to different results depending on the random seed. In relatively easy cases as the synthetic data with two components, where all the data can be explained perfectly, the randomness has little effect (though it might change the order or scale of the components). In more complex situations, there might be more drastic changes.

Applying NMF to face images

Now, let's apply NMF to the "labeled faces in the wild" dataset we used above.

The main parameter of NMF is how many components we want to extract. Usually this is lower than the number of input features (otherwise the data could be explained by making each pixel a separate component).

First, let's inspect how the number of components impacts how well the data can be reconstructed using NMF:

```
mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
```


```
_________________________________________________________________________

[Memory] Calling mglearn.plot_nmf.nmf_faces...
```

```
nmf_faces(array([[ 0.036601, ...,  0.742484],

       ...,

       [ 0.105882, ...,  0.393464]], dtype=float32),

array([[ 0.162091, ...,  0.677124],

       ...,

       [ 0.109804, ...,  0.07451 ]], dtype=float32))

_____________________________________________________nmf_faces - 763.1s, 12.7min
```

The quality of the back-transformed data is similar to PCA, but slightly worse. This is expected, as PCA finds the optimum directions in terms of reconstruction. NMF is usually not used for the ability to reconstruct or encode data, but rather for finding interesting patterns within the data.

As a first look into the data, let's try extracting only a few components, say 15, on the faced data:

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=15, random_state=0)
nmf.fit(X_train)
X_train_nmf = nmf.transform(X_train)
X_test_nmf = nmf.transform(X_test)

fix, axes = plt.subplots(3, 5, figsize=(15, 12),
                         subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape))
    ax.set_title("%d. component" % i)
```

These components are all positive, and so resemble prototypes of faces much more so than the components shown for PCA in Figure Eigenfaces. For example, one can clearly see that component 3 shows a face rotated somewhat to the right, while component 7 shows a face somewhat rotated to the left. Let's look at the images for which these components are particularly strong:

```
compn = 3
# sort by 3rd component, plot first 10 images
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("Large component 3")
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))

compn = 7
# sort by 7th component, plot first 10 images
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig.suptitle("Large component 7")
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
```

```
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))
```

As expected, faces that have a high coefficient for component 3 are faces looking to the right, while faces with a high component 7 are looking to the left. As mentioned above, extracting patterns like these works best for data with additive structure, including audio, gene expression data, and text data. We will see applications of NMF to text data in Chapter 7 (Text Data).

There are many other algorithms that can be used to decompose each data point into a weighted sum of a fixed set of components, as PCA and NMF do. Discussing all of them is beyond the scope of this book, and describing the constraints made on the components and coefficients often involves probability theory. If you are interested in these kinds of pattern extraction, we recommend to study the user guide of Independent Component Analysis (ICA), Factor Analysis (FA) and Sparse Coding (dictionary learning), which are widely used decomposition methods.

Manifold learning with t-SNE

While PCA is often a good first approach for transforming your data so that you might be able to visualize it using a scatter plot, the nature of the method (applying a rotation and then dropping directions) limits its usefulness, as we saw with the scatter plot of the labeled faces in the wild. There is a class of algorithms for visualization called *manifold learning* algorithms which allows for much more complex mappings, and often provides better visualizations. A particular useful one is the t-SNE algorithm.

Manifold learning algorithms are mainly aimed at visualization, and so are rarely used to to generate more than two new features. Some of them, including t-SNE,

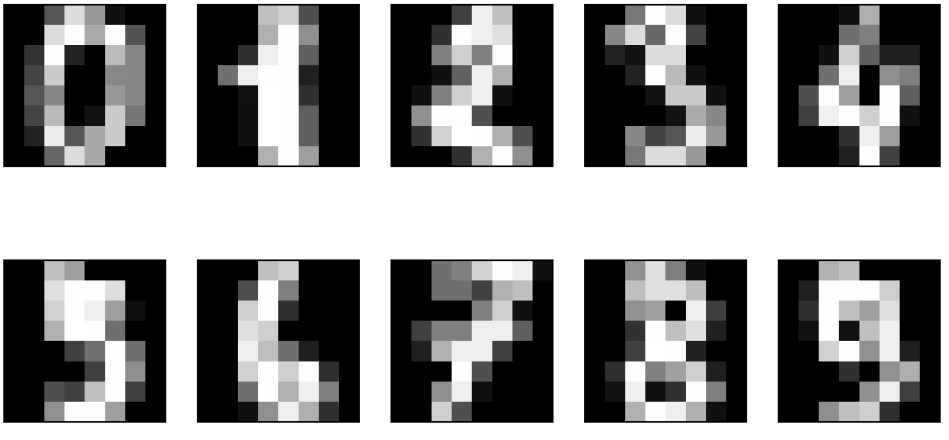
compute a new representation of the training data, but don't allow transformations of new data. This means these algorithms can not be applied to a test set: rather, they can only transform the data they were trained for. Manifold learning can be useful for exploratory data analysis, but is rarely used if the final goal is supervised learning.

The idea behind t-SNE is to find a two-dimensional representation of the data that preserve the distances between points as best as possible. t-SNE tarts with a random two-dimensional representation for each data point, and then tries to make points closer that are close in the original feature space, and points far apart that are far apart in the original feature space. t-SNE puts more emphasis on points that are close by, rather than preserving distances between far apart points. In other words, it tries to preserve the information of which points are neighbors to each other.

We will apply the t-SNE manifold learning algorithm on a dataset of handwritten digits that is included in scikit-learn [Footnote: not to be confused with the much larger MNIST dataset].

Each data point in this dataset is a 8x8 grey-scale image of a handwritten digit between 0 and 1. Here is an example image for each class:

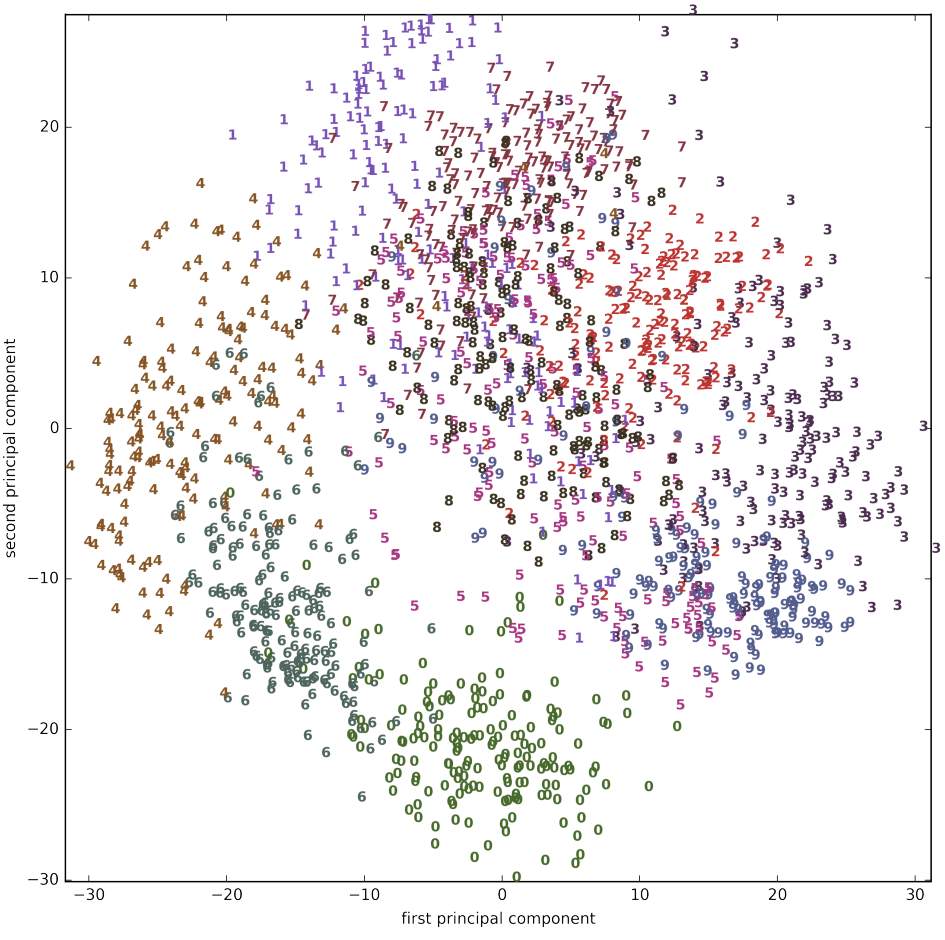
```
from sklearn.datasets import load_digits
digits = load_digits()

fig, axes = plt.subplots(2, 5, figsize=(10, 5),
                         subplot_kw={'xticks':(), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
    ax.imshow(img)
```



Let's use PCA to visualize the data reduced to two dimensions. We plot the first two principal components, and color each dot by its class:

```
# build a PCA model
pca = PCA(n_components=2)
```

```
pca.fit(digits.data)
# transform the digits data onto the first two principal components
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120","#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max())
plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max())
for i in range(len(digits.data)):
    # actually plot the digits as text instead of using scatter
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("first principal component")
plt.ylabel("second principal component")
```

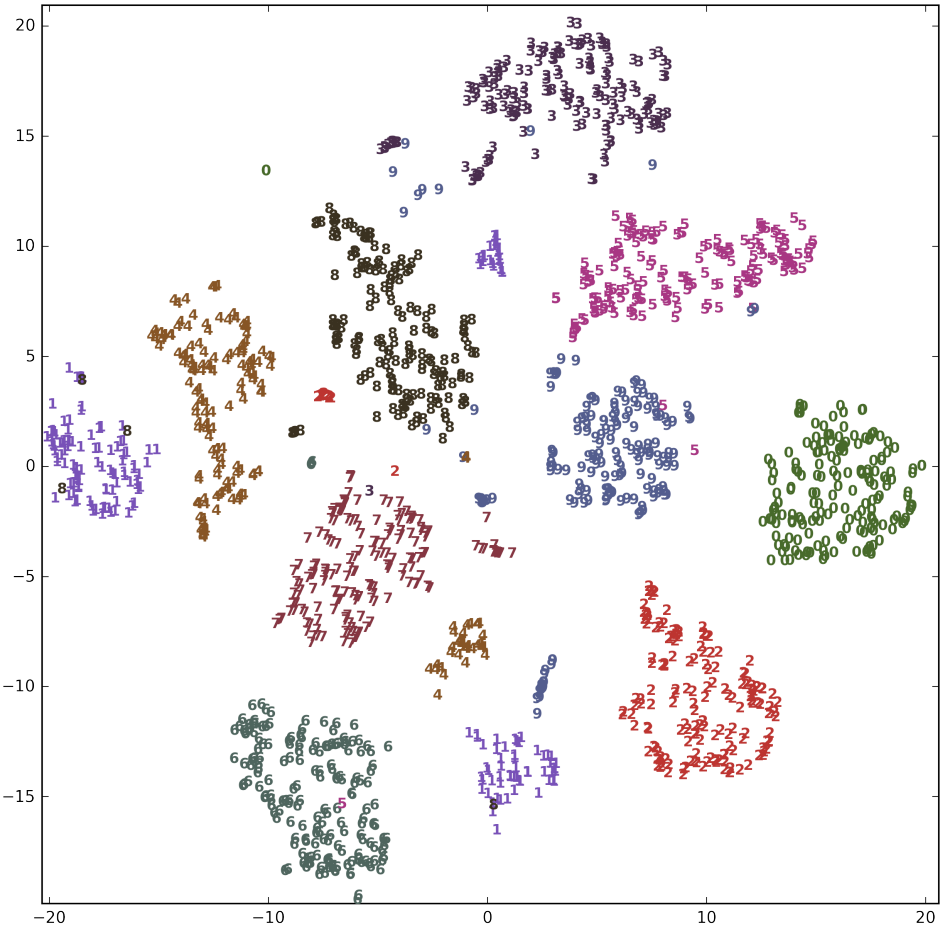


Here, we actually used the true digit classes as glyphs, to show which class is where. The digits zero, six and four are relatively well-separated using the first two principal components, though they still overlap. Most of the other digits overlap significantly.

Let's apply t-SNE to the same dataset, and compare results. As t-SNE does not support transforming new data, the `TSNE` class has no `transform` method. Instead, we can call the `fit_transform` method, which will build the model, and immediately return the transformed data:

```
from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)
# use fit_transform instead of fit, as TSNE has no transform method:
digits_tsne = tsne.fit_transform(digits.data)
```

```
plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(), digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
    # actually plot the digits as text instead of using scatter
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
```



The result of t-SNE is quite remarkable. All the classes are quite clearly separated. The ones and nines are somewhat split up, but most of the classes form a single dense group. Keep in mind that this method has no knowledge of the class labels: it is completely unsupervised. Still, it can find a representation of the data in two dimensions that clearly separates the classes, based solely on how close points are in the original space.

t-SNE has some tuning parameters, though it often works well with the default settings. You can try playing with `perplexity` and `early_exaggeration`, though the effects are usually minor.

Clustering

As we described above, clustering is the task of partitioning the dataset into groups, called clusters. The goal is to split up the data in such a way that points within a single cluster are very similar and points in different clusters are different. Similarly to classification algorithms, clustering algorithms assign (or predict) a number to each data point, indicating which cluster a particular point belongs to.

k-Means clustering

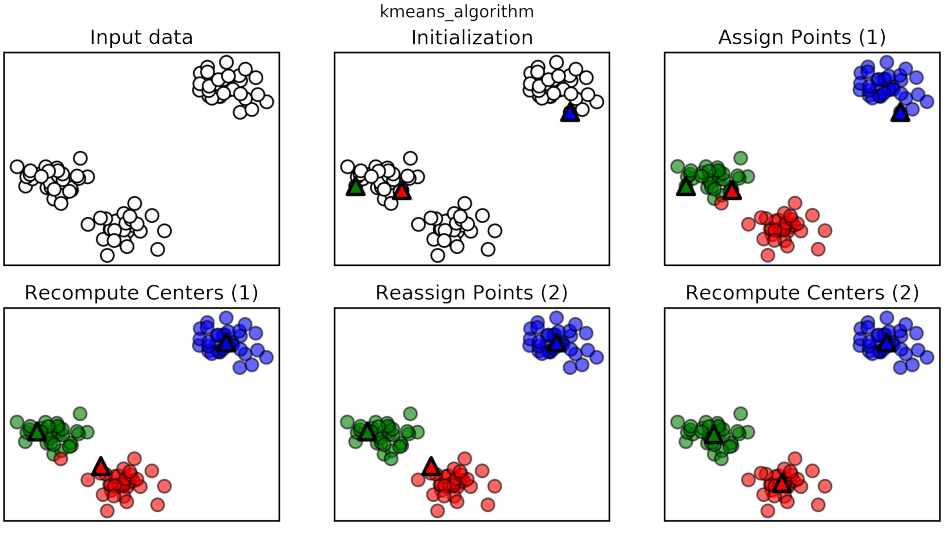
k-Means clustering is one of the simplest and most commonly used clustering algorithms. It tries to find *cluster centers* that are representative of certain regions of the data.

The algorithm alternates between two steps: assigning each data point to the closest cluster center, and then setting each cluster center as the mean of the data points that are assigned to it.

The algorithm is finished when the assignment of instances to clusters no longer changes.

Figure kmeans_algorithm illustrates the algorithm on a synthetic dataset:

```
mglearn.plots.plot_kmeans_algorithm()
plt.suptitle("kmeans_algorithm");
```

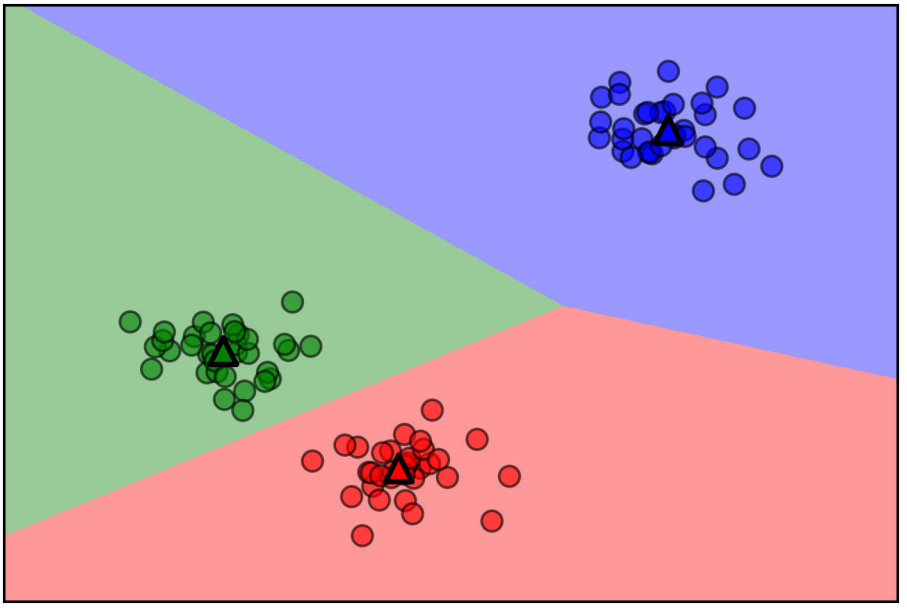


We specified that we are looking for three clusters, so the algorithm was initialized by declaring three data points as cluster centers (see "Initialization"). Then the iterative algorithm starts: Each data point is assigned to the cluster center it is closest to (see

"Assign Points (1)"). Next, the cluster centers are updated to be the mean of the assigned points (see "Recompute Centers (1)"). Then the process is repeated. After the second iteration, the assignment of points to cluster centers remained unchanged, so the algorithm stops.

Given new data points, k-Means will assign them to the closest cluster center. Here are the boundaries of the cluster centers that were learned in the diagram above:

```
mglearn.plots.plot_kmeans_boundaries()
```



Applying k-Means with scikit-learn is quite straight-forward. Here we apply it to the synthetic data that we used for the plots above. We instantiate the `KMeans` class, and set the number of clusters we are looking for [footnote: If you don't provide `n_clusters` it is set to eight by default. There is no particular reason why you should use eight.]. Then we call the `fit` method with the data:

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

# generate synthetic two-dimensional data
X, y = make_blobs(random_state=1)

# build the clustering model:
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```

```
KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=3, n_init=10,
    n_jobs=1, precompute_distances='auto', random_state=None, tol=0.0001,
    verbose=0)
```

During the algorithm, each training data point in `X` is assigned a cluster label. You can find these labels in the `kmeans.labels_` attribute:

```
print(kmeans.labels_)
```

```
[1 9 3 6 9 4 1 2 1 2 7 8 9 2 0 1 5 8 5 0 4 2 5 2 7 3 9 4 3 5 1 7 2 3 1 4 8
 1 0 3 8 7 5 3 9 7 1 0 3 2 0 8 4 6 2 1 5 2 5 7 8 6 9 1 2 6 7 9 7 6 8 6 8 3
 2 6 3 1 5 8 4 7 8 4 3 7 1 7 8 4 5 1 4 0 4 9 9 8 6 3 2 4 7 1 6 3 6 7 9 5 0
 7 7 6 1 9 5 4 8 1 1 0 3 7 3 0 1 3 2 4 1 0 6 8 2 9 2 6 4 1 3 3 5 7 7 1 3 2
 5 3 8 9 1 5 8 7 2 8 5 5 3 2 5 9 8 9 5 8 9 2 5 6 6 0 3 2 4 0 0 5 9 6 4 9 4
 5 7 2 6 0 6 5 1 1 3 0 4 3 5 9]
```

As we asked for three clusters, the clusters are numbered 0 to 2.

You can also assign cluster labels to new points, using the `predict` method. Each new point is assigned to the closest cluster center when predicting, but the existing model is not changed. Running `predict` on the training set returns the same as `labels_`:

```
print(kmeans.predict(X))
```

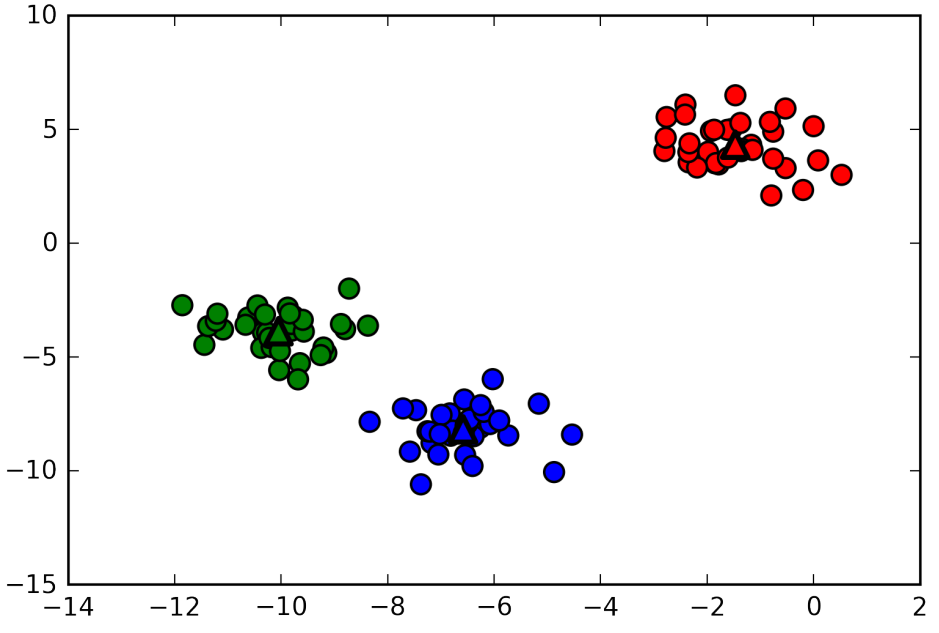
```
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
 2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
 1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

You can see thgat clustering is somewhat similar to classification, in that each item gets a label. However, there is no ground truth, and consequently the labels themselves have no *a priori* meaning. Let's go back to the example of clustering face images that we discusse before. It might be that the cluster 3 found by the algorithm contains only faces of your friend Bela. You can only know that after you looked at the pictures, though, and the number 3 is arbitrary. The only information the algorithm gives you is that all faces labeled as 3 are similar.

For the clustering we just computed on the two dimensional toy dataset, that means that

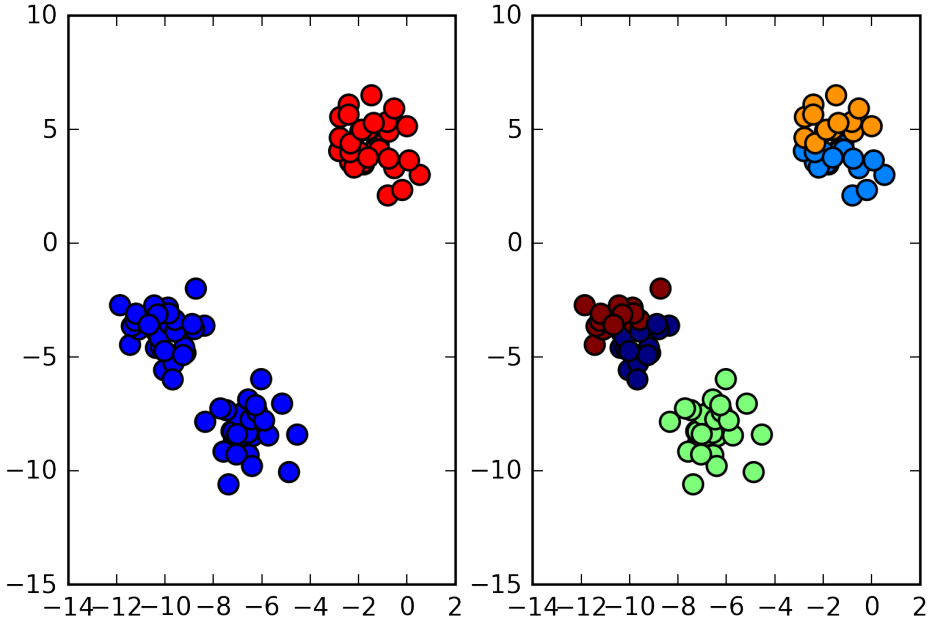
Here is a plot of this data again. The cluster centers are stored in the `cluster_centers_` attribute, and we plot them as triangles:

```
plt.scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap=mglearn.cm3, s=60)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', s=100, linewidth=2, c=[0, 1, 2], cmap=mglearn.cm3)
```



We can also use more or less cluster centers:

```
fig, axes = plt.subplots(1, 2)

# using two cluster centers:
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
assignments = kmeans.labels_

axes[0].scatter(X[:, 0], X[:, 1], c=assignments, cmap=mglearn.cm2, s=60)

# using five cluster centers:
kmeans = KMeans(n_clusters=5)
kmeans.fit(X)
assignments = kmeans.labels_

axes[1].scatter(X[:, 0], X[:, 1], c=assignments, cmap='jet', s=60);
```

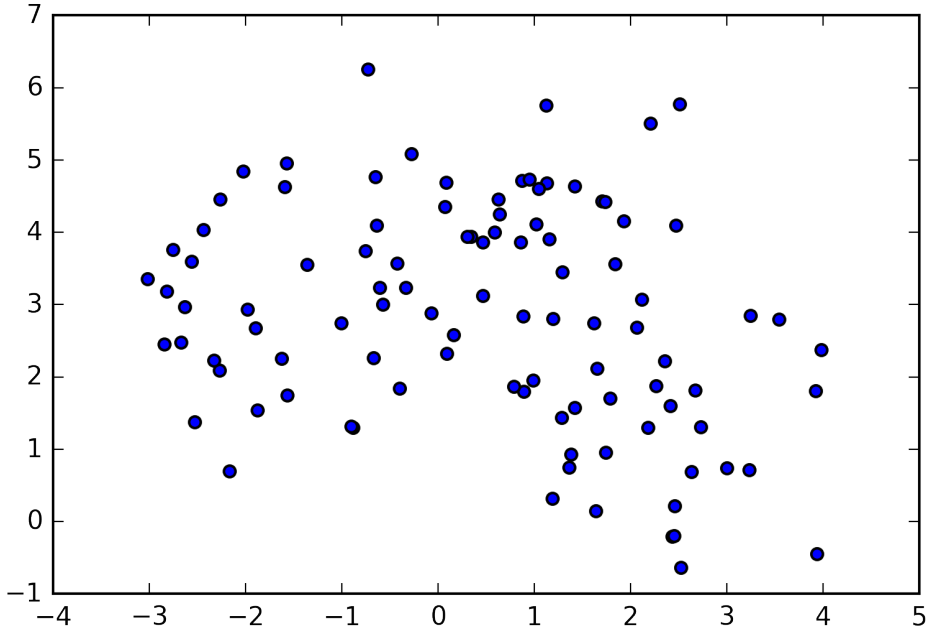


Failure cases of k-Means

Even if you know the "right" number of clusters for a given dataset, k-Means might not always be able to recover them. Each cluster is defined solely by its center, which means that each cluster is a convex shape. As a result of this is that k-Means can only capture relatively simple shapes.

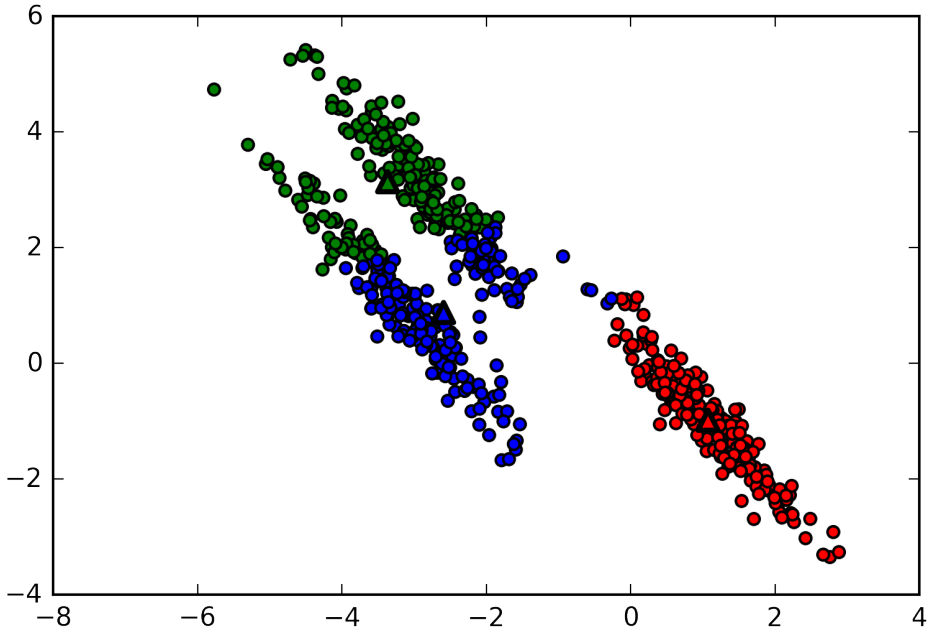
k-Means also assumes that all clusters have the same "diameter" in some sense; it always draws the boundary between clusters to be exactly in the middle between the cluster centers. That can sometimes lead to surprising results, as shown below:

```
X, y = make_blobs(random_state=0)
plt.scatter(X[:, 0], X[:, 1]);
```



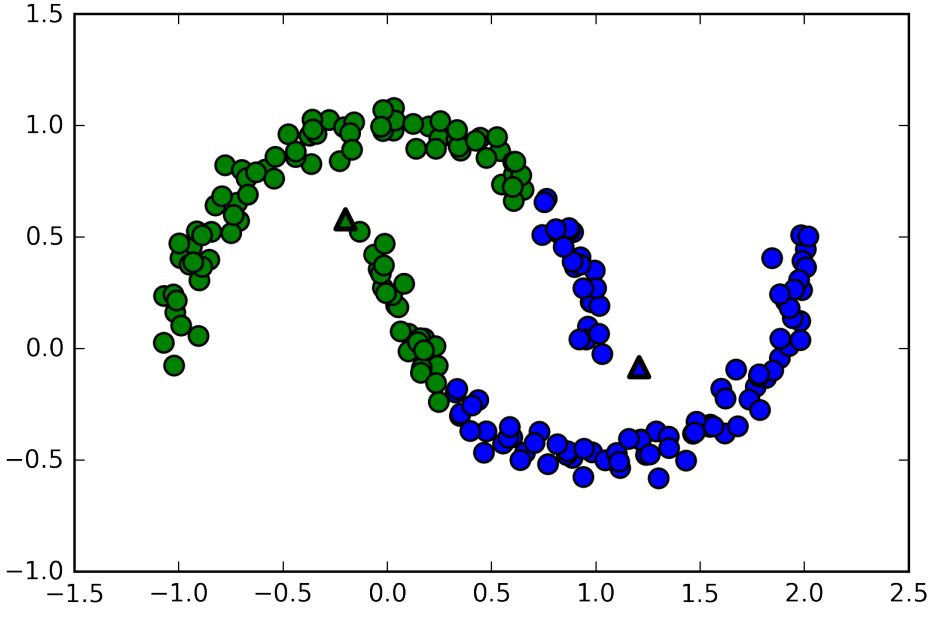
k-Means also assumes that all directions are equally important for each cluster. The plot below shows a two-dimensional dataset where there are three clearly separated parts in the data. However, these groups are stretched towards the diagonal. As k-Means only considers the distance to the nearest cluster center, it can't handle this kind of data.

```
# generate some random cluster data
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)

# transform the data to be streched
transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)

# cluster the data into three clusters
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)

# plot the cluster assignments and cluster centers
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm3)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', c=['b', 'r', 'g'], s=60, linewidth=2);
```



k-Means also performs poorly if the clusters have more complex shapes, like the `two_moons` data we encountered in Chapter 2:

```
# generate synthetic two_moons data (with less noise this time)
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# cluster the data into two clusters
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_pred = kmeans.predict(X)

# plot the cluster assignments and cluster centers
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm3, s=60)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', c=['b', 'g'], s=60, linewidth=2);
```



Here, we would hope that the clustering algorithm can discover the two half-moon shapes. However, this is not possible using the k-Means algorithm.

Vector Quantization - Or Seeing k-Means as Decomposition

Even though k-Means is a clustering algorithm, there are interesting parallels between k-Means and decomposition methods like PCA and NMF that we discussed above. You might remember that PCA tries to find directions of maximum variance in the data, while NMF tries to find additive components, which often correspond to "extremes" or "parts" of the data (see Figure nmf_illustration). Both methods tried to express data points as a sum over some components.

k-Means on the other hand tries to represent each data point using a cluster center. You can think of that as each point being represented using only a single component, which is given by the cluster center. This view of k-Means as a decomposition method, where each point is represented using a single component, is called *vector quantization*.

Here is a side-by-side comparison of PCA, NMF and k-Means, showing the components extracted, as well as reconstructions of faces from the test set using 100 components. For k-Means, the reconstruction is the closest cluster center found on the training set:

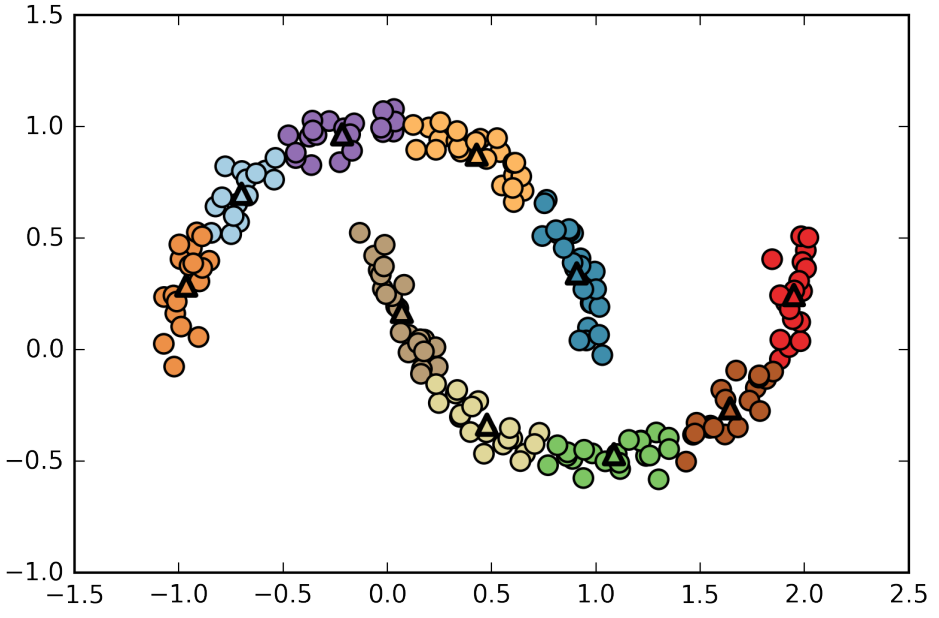
```
X_train, X_test, y_train, y_test = train_test_split(X_people, y_people, stratify=y_people, random_
nmf = NMF(n_components=100)
```

```
nmf.fit(X_train)
pca = PCA(n_components=100)
pca.fit(X_train)
kmeans = KMeans(n_clusters=100)
kmeans.fit(X_train)

X_reconstructed_pca = pca.inverse_transform(pca.transform(X_test))
X_reconstructed_kmeans = kmeans.cluster_centers_[kmeans.predict(X_test)]
X_reconstructed_nmf = np.dot(nmf.transform(X_test), nmf.components_)

fig, axes = plt.subplots(3, 5, figsize=(8, 8)) #, subplot_kw={'xticks': (), 'yticks': ()}
fig.suptitle("Extracted Components")
for ax, comp_kmeans, comp_pca, comp_nmf in zip(axes.T, kmeans.cluster_centers_, pca.components_, r
    ax[0].imshow(comp_kmeans.reshape(image_shape))
    ax[1].imshow(comp_pca.reshape(image_shape), cmap='viridis')
    ax[2].imshow(comp_nmf.reshape(image_shape))

axes[0, 0].set_ylabel("kmeans")
axes[1, 0].set_ylabel("pca")
axes[2, 0].set_ylabel("nmf")

fig, axes = plt.subplots(4, 5, subplot_kw={'xticks': (), 'yticks': ()}, figsize=(8, 8))
fig.suptitle("Reconstructions")
for ax, orig, rec_kmeans, rec_pca, rec_nmf in zip(axes.T, X_test, X_reconstructed_kmeans,
                                                  X_reconstructed_pca, X_reconstructed_nmf):
    ax[0].imshow(orig.reshape(image_shape))
    ax[1].imshow(rec_kmeans.reshape(image_shape))
    ax[2].imshow(rec_pca.reshape(image_shape))
    ax[3].imshow(rec_nmf.reshape(image_shape))

axes[0, 0].set_ylabel("original")
axes[1, 0].set_ylabel("kmeans")
axes[2, 0].set_ylabel("pca")
axes[3, 0].set_ylabel("nmf")
```

An interesting aspect of vector quantization using k-Means is that we can use many more clusters than input dimensions to encode our data. Let's go back to the `two_moons` data. Using PCA or NMF, there is nothing much we can do to this data, as it lives in only two dimensions. Reducing it to one dimension with PCA or NMF would completely destroy the structure of the data. But we can find a more expressive representation using k-Means, by using more cluster centers:

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

kmeans = KMeans(n_clusters=10)
kmeans.fit(X)
y_pred = kmeans.predict(X)
```

```
plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=60, cmap='Paired')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', c=range(kmeans.n_clusters), s=60, linewidth=2, cmap='Paired')
print(y_pred)
```




```
[1 9 3 6 9 4 1 2 1 2 7 8 9 2 0 1 5 8 5 0 4 2 5 2 7 3 9 4 3 5 1 7 2 3 1 4 8
 1 0 3 8 7 5 3 9 7 1 0 3 2 0 8 4 6 2 1 5 2 5 7 8 6 9 1 2 6 7 9 7 6 8 6 8 3
 2 6 3 1 5 8 4 7 8 4 3 7 1 7 8 4 5 1 4 0 4 9 9 8 6 3 2 4 7 1 6 3 6 7 9 5 0
 7 7 6 1 9 5 4 8 1 1 0 3 7 3 0 1 3 2 4 1 0 6 8 2 9 2 6 4 1 3 3 5 7 7 1 3 2
 5 3 8 9 1 5 8 7 2 8 5 5 3 2 5 9 8 9 5 8 9 2 5 6 6 0 3 2 4 0 0 5 9 6 4 9 4
 5 7 2 6 0 6 5 1 1 3 0 4 3 5 9]
```

We used 10 cluster centers, which means each point is now assigned a number between 0 and 9. We can see this as the data being represented using 10 components (that is, we have ten new features), with all features being zero, apart from the one that represents the cluster center the point is assigned to. Using this 10-dimensional representation, it would now be possible to separate the two half-moon shapes using a linear model, which would not have been possible using the original two features. [really? can you show what that would look like, maybe?]

It is also possible to get an even more expressive representation of the data by using the distances to each of the cluster centers as features. This can be done using the `transform` method of `kmeans`:

```
distance_features = kmeans.transform(X)
print(distance_features.shape)
print(distance_features)
```

```
(200, 10)

[[ 1.53  0.2   1.03 ...,  1.12  0.92  1.14]

 [ 2.56  1.01  0.54 ...,  2.28  1.14  0.12]

 [ 0.8   0.93  1.33 ...,  0.72  0.79  1.75]

 ...,

 [ 1.12  0.81  1.02 ...,  1.05  0.45  1.49]

 [ 0.88  1.03  1.76 ...,  0.34  1.39  1.98]

 [ 2.5   0.91  0.59 ...,  2.19  1.15  0.05]]
```

k-Means is a very popular algorithm for clustering, not only because it is relatively easy to understand and implement, but also because it runs relatively quickly. k-Means scales easily to large datasets, and scikit-learn even includes a more scalable variant in the `MiniBatchKMeans` class, which can handle very large datasets.

One of the drawbacks of k-Means is that it relies on a random initialization, which means the outcome of the algorithm depends on a random seed. By default, scikit-learn runs the algorithm 10 times with 10 different random initializations, and returns the best result [Footnote: best here meaning that the sum of variances of the clusters is small].

Further downsides of k-Means are the relatively restrictive assumptions made on the shape of clusters, and the requirement to specify the number of clusters you are looking for (which might not be known in a real-world application).

Next, we will look at two more clustering algorithms that improve upon these properties in some ways.

Agglomerative Clustering

Agglomerative clustering refers to a collection of clustering algorithms that all build upon the same principles: The algorithm starts by declaring each point its own cluster, and then merges the two most similar clusters until some stopping criterion is satisfied.

The stopping criterion implemented in scikit-learn is the number of clusters, so similar cluster are merged until only the specified number of clusters is left.

There are several *linkage* criteria that specify how exactly "most similar cluster" is measured. [this sentence looks a lot like the next sentence - combine into one]

The following three choices are implemented in scikit-learn:

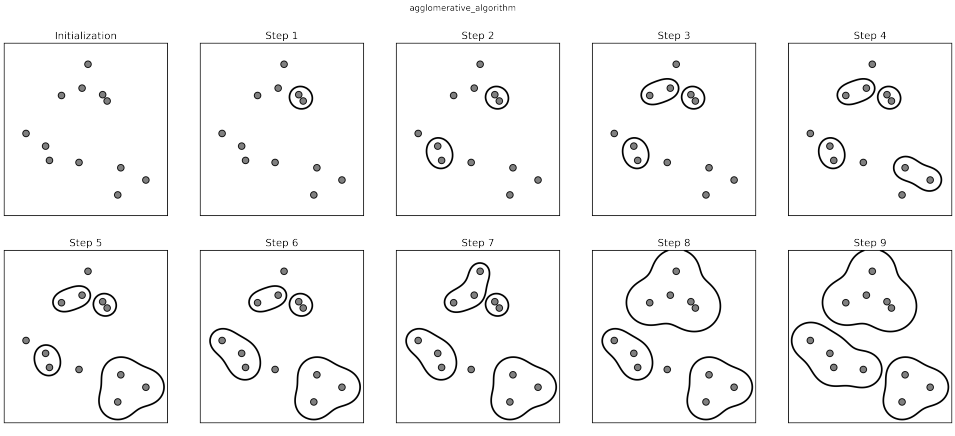
- "ward", which is the default choice. Ward picks the two clusters to merge such that the variance within all clusters increases the least. This often leads to clusters that are relatively equally sized.
- "average" linkage merges the two clusters that have the smallest average distance between all their points.
- "complete" linkage (also known as maximum linkage) merges the two clusters that have the smallest maximum distance between their points.

[you keep writing 'the two clusters', but you did not specify that there would be two clusters - rahter said that there are a 'number of clusters']

Ward is generally a good default; all our examples below will use ward.

The plot below illustrates the progression of agglomerative clustering on a two-dimensional dataset, looking for three clusters.

```
mglearn.plots.plot_agglomerative_algorithm()
plt.suptitle("agglomerative_algorithm");
```



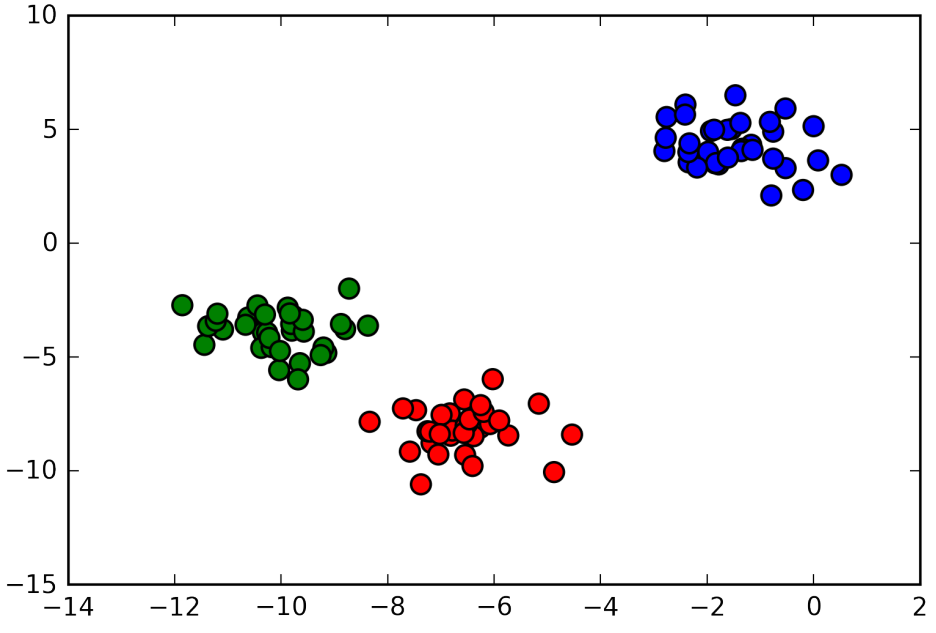
In the beginning, each point is its own cluster. Then, in each step, the two clusters that are closest are merged. In the first four steps, two single point clusters are picked and these are joined into two-point clusters. In step four, one of the two-point clusters is extended to a third point, and so on. In step 9, there are only three clusters

remaining. As we specified that we are looking for three clusters, the algorithm then stops.

Let's have a look at how agglomerative clustering performs on the simple three-cluster data we used above.

Because of the way the algorithm works, agglomerative clustering can not make predictions for new data points. Therefore, agglomerative clustering has no `predict` method. To build the model, and get the cluster memberships on the training set, use the `fit_predict` method instead. [footnote: we could also use the `labels_` attribute as we did for k-Means]

```
from sklearn.cluster import AgglomerativeClustering
X, y = make_blobs(random_state=1)

agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)

plt.scatter(X[:, 0], X[:, 1], c=assignment, cmap=mglearn.cm3, s=60)
```



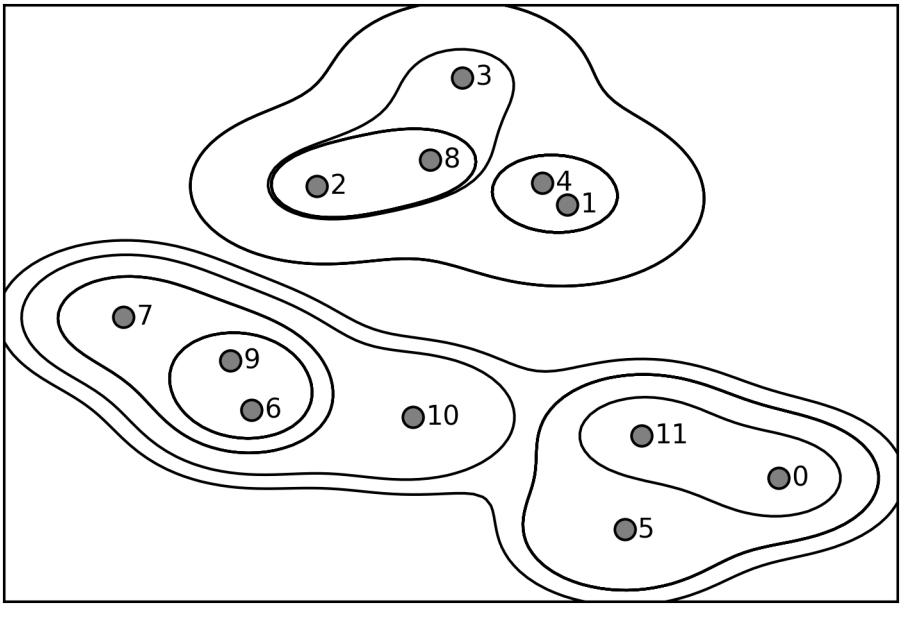
As expected, the algorithm recovers the clustering perfectly. While the scikit-learn implementation of agglomerative clustering requires you to specify a number of clusters you want the algorithm to find, agglomerative clustering methods provide some help with choosing the right number, which we will now discuss next.

Hierarchical Clustering and Dendrograms

Agglomerative clustering produces what is known as a *hierarchical clustering*. The clustering proceeds iteratively, and every point makes a journey from being a single point cluster to belonging to some final cluster. Each intermediate step provides a clustering of the data (with a different number of clusters). It is sometimes helpful to look at all possible clusterings jointly.

The figure below shows an overlay of all possible clusterings shown in Figure agglomerative_algorithm, providing some insight into how each cluster breaks up into smaller clusters.

```
mglearn.plots.plot_agglomerative()
```



While this visualization provides a very detailed view of the hierarchical clustering, it relies on the two-dimensional nature of the data, and can therefore not be used on datasets that have more than two features. There is, however, another tool to visualize hierarchical clustering, called a *dendrogram* (as shown in Figure dendrogram below).

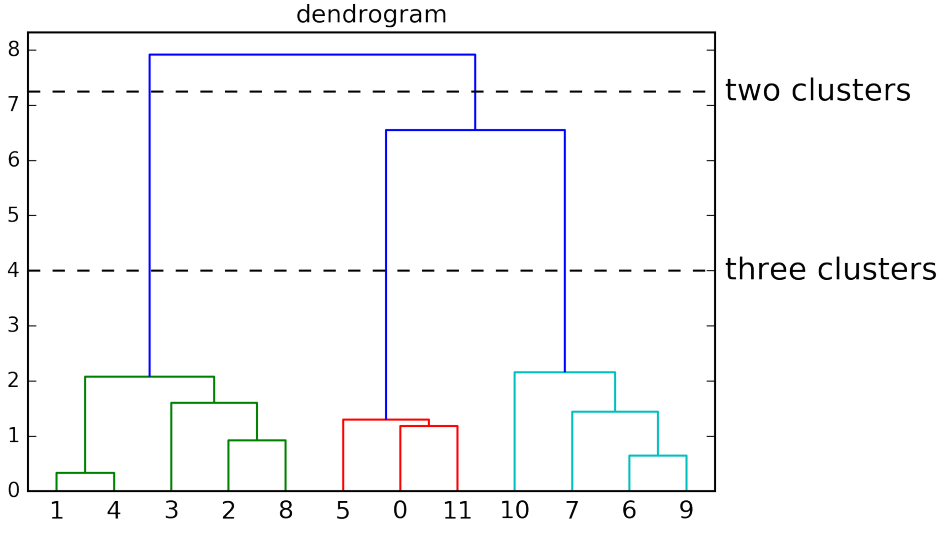
Unfortunately, scikit-learn currently does not have the functionality to draw dendrograms. However, you can generate them easily using `scipy`.

The scipy clustering algorithms have a slightly different interface from the scikit-learn clustering algorithms.

scipy provides function that take data arrays `X` *linkage array* encoding cluster similarities.

We can then feed this linkage array into the scipy `dendrogram` function to plot the dendrogram:

```
# import the dendrogram function and the ward clustering function from scipy
from scipy.cluster.hierarchy import dendrogram, ward

X, y = make_blobs(random_state=0, n_samples=12)
# apply the ward clustering to the data array X
# The scipy ward function returns an array that specifies the distances bridged when performing ag
linkage_array = ward(X)
# now we plot the dendrogram for the linkage_array containing the distances between clusters
dendrogram(linkage_array);

# mark the cuts in the tree that signify two or three clusters
ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')

ax.text(bounds[1], 7.25, ' two clusters', verticalalignment='center', fontdict={'size': 15})
ax.text(bounds[1], 4, ' three clusters', verticalalignment='center', fontdict={'size': 15})
plt.title("dendrogram")
```



The dendrogram shows data points as points on the bottom (numbered from zero to eleven). Then, a tree is plotted with these points (representing single-point clusters) as the leafs, and a new node parent is added for each two clusters that are joined.

Reading from bottom to top, the data points 1 and 4 are joined first (as you could see in Figure agglomerative_algorithm). Next, points 6 and 9 are joined into a cluster, and so on. The top level, there are two branches, one consisting of point 11, 0, 5, 10,

7, 6 and 9, and the other one consisting of points 1, 4, 3, 2 and 8. These correspond to the two largest clusters in in the left hand side of the plot.

The y axis in the dendrogram not only specifies when in the agglomerative algorithm two clusters get merged. The length of each branch also shows how far apart the merged clusters are. The longest branches in this dendogram are the three lines that are around the "10" mark on the y-axis. That these are the longest branches indicates that going from three to two clusters meant merging some very far-apart points. We see this again at the top of the chart, where merging the two remaining clusters into a single cluster again bridges a large distance.

Unfortunately, agglomerative clustering still fails at separating complex shapes like the `two_moons` dataset. The same is not true for the next algorithm we will look at, DBSCAN.

DBSCAN

Another very useful clustering algorithm is DBSCAN (which stands for "Density-based spatial clustering of applications with noise"). The main benefits of DBSCAN are that a) it does not require the user to set the number of clusters *a priori*, b) it can capture clusters of complex shapes, and c) it can identify point that are not part of any cluster.

DBSCAN is somewhat slower than agglomerative clustering and k-Means, but still scales to relatively large datasets.

The way DBSCAN works is by identifying points that are in "crowded" regions of the feature space, where many data points are close together. These regions are referred to as *dense* regions in feature space. The idea behind DBSCAN is that clusters form dense regions of data, separated by regions that are relatively empty.

Points that are within a dense region are called *core samples*, and they are defined as follows.

There are two parameters in DBSCAN, `min_samples` and `eps`. If there are at least `min_samples` many data points within a distance of `eps` to a given data point, it's called a *core sample*. Core samples that are closer than the distance `eps` are put into the same cluster by DBSCAN.

The algorithm works by picking a point to start with.

It then finds all points with distance `eps` or less. If there are less than `min_samples` points within distance `eps` or less, this point is labeled as *noise*, meaning that this point doesn't belong to any cluster.

If there are more than `min_samples` points within a distance of `eps`, the point is labeled a core sample and assigned a new cluster label. Then, all neighbors (withing

eps) of the point are visited. If they have not been assigned a cluster yet, they are assigned the new cluster label we just created. If they are core samples, their neighbors are visited in turn, and so on.

The cluster grows, until there are no more core-samples within distance eps of the cluster.

Then another point, which hasn't yet been visited, is picked, and the same procedure is repeated.

In the end, there are three kinds of points: core points, points that are within distance eps of core points (called boundary points), and noise. When running the DBSCAN algorithm on a particular dataset multiple times, the clustering of the core points is always the same, and the same points will always be labeled as noise. However, a boundary point might be neighbor to core samples of more than one cluster. Therefore, the cluster membership of boundary points depends on the order in which points are visited. Usually there are only few boundary points, and this slight dependence on the order of points is not important.

Let's apply DBSCAN on the synthetic data from above. As in agglomerative clustering, DBSCAN does not allow predictions on new test data, so we will use the fit_predict method to perform clustering and return the cluster labels in one step:

```
from sklearn.cluster import DBSCAN
X, y = make_blobs(random_state=0, n_samples=12)

dbscan = DBSCAN()
clusters = dbscan.fit_predict(X)
clusters
```

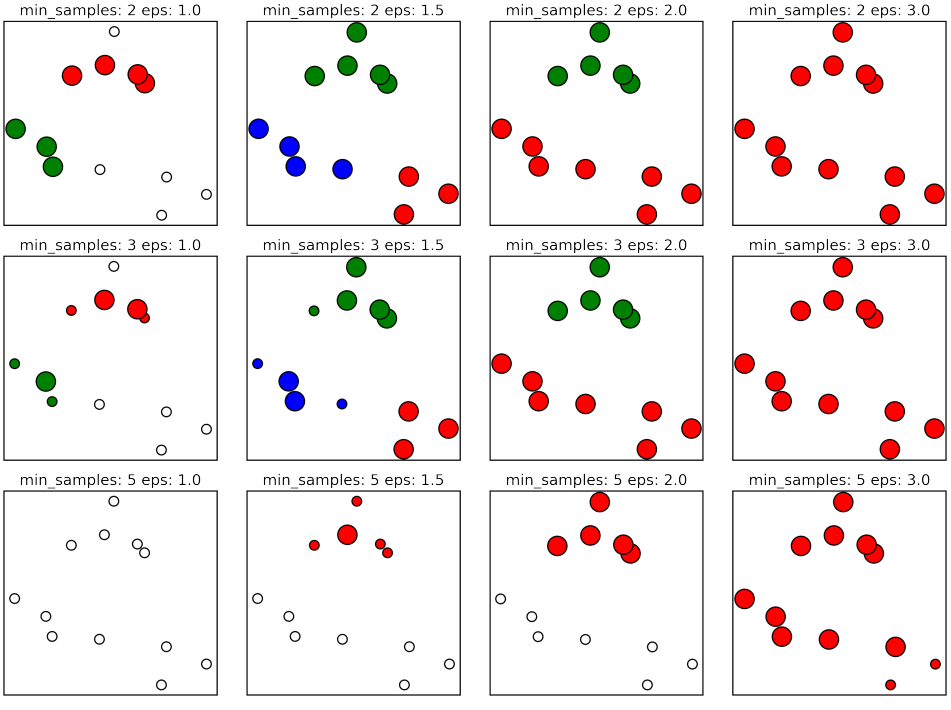
```
array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1])
```

As you can see, all data points were assigned the label -1, which stands for noise. This is a consequence of the default parameter settings for eps and min_samples, which are not attuned to small toy datasets.

Let's investigate the effect of changing eps and min_samples.

```
fig, axes = plt.subplots(3, 4, figsize=(11, 8), subplot_kw={'xticks': (), 'yticks': ()})
# Plot clusters as red, green and blue, and outliers (-1) as white
colors = np.array(['r', 'g', 'b', 'w'])

# iterate over settings of min_samples and eps
for i, min_samples in enumerate([2, 3, 5]):
    for j, eps in enumerate([1, 1.5, 2, 3]):
        # instantiate DBSCAN with a particular setting
        dbscan = DBSCAN(min_samples=min_samples, eps=eps)
        # get cluster assignments
        clusters = dbscan.fit_predict(X)
        print("min_samples: %d eps: %f  cluster: %s" % (min_samples, eps, clusters))
```

```
        # vizualize core samples and clusters.
        sizes = 60 * np.ones(X.shape[0])
        # size is given by whether something is a core sample
        sizes[dbscan.core_sample_indices_] *= 4
        axes[i, j].scatter(X[:, 0], X[:, 1], c=colors[clusters], s=sizes)
        axes[i, j].set_title("min_samples: %d eps: %.1f" % (min_samples, eps))
fig.tight_layout()
```




```
min_samples: 2 eps: 1.000000  cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1 -1]

min_samples: 2 eps: 1.500000  cluster: [0 1 1 1 1 0 2 2 1 2 2 0]

min_samples: 2 eps: 2.000000  cluster: [0 1 1 1 1 0 0 0 1 0 0 0]

min_samples: 2 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]

min_samples: 3 eps: 1.000000  cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1 -1]

min_samples: 3 eps: 1.500000  cluster: [0 1 1 1 1 0 2 2 1 2 2 0]

min_samples: 3 eps: 2.000000  cluster: [0 1 1 1 1 0 0 0 1 0 0 0]

min_samples: 3 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]

min_samples: 5 eps: 1.000000  cluster: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

```
min_samples: 5 eps: 1.500000  cluster: [-1  0  0  0  0 -1 -1 -1  0 -1 -1 -1]

min_samples: 5 eps: 2.000000  cluster: [-1  0  0  0  0 -1 -1 -1  0 -1 -1 -1]

min_samples: 5 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
```

In this plot, points that belong to clusters are colored, while the noise points are shown in white.

Core samples are shown as large points, while border points are displayed as smaller points.

Increasing `eps` (going from left to left to right in the figure) means that more points will be included in a cluster. This makes clusters grow, but might also lead to multiple clusters joining into one. Increasing `min_samples` (going from top to bottom in the figure) means that fewer points will be core points, and more points will be labeled as noise.

The parameter `eps` is somewhat more important, as it determines what it means for points to be "close".

Setting `eps` to be very small will mean that no points are core samples, and may lead to all points being labeled as noise. Setting `eps` to be very large will result in all points forming a single cluster.

The setting of `min_samples` mostly determines whether points in less dense regions will be labeled as outliers, or as their own cluster. If you decrease `min_samples`, anything that would have been a cluster with less than `min_samples` many samples will now be labeled as noise. The `min_samples` therefore determines the minimum cluster size. You can see this very clearly in the plot above, when going from `min_samples=3` to `min_samples=5` with `eps=1.5`. With `min_samples=3`, there are three clusters: one of four points, one of five points and one of three points. Using `min_samples=5`, the two smaller clusters (with three or four points) are now labeled as noise, and only the cluster with 5 samples remains.

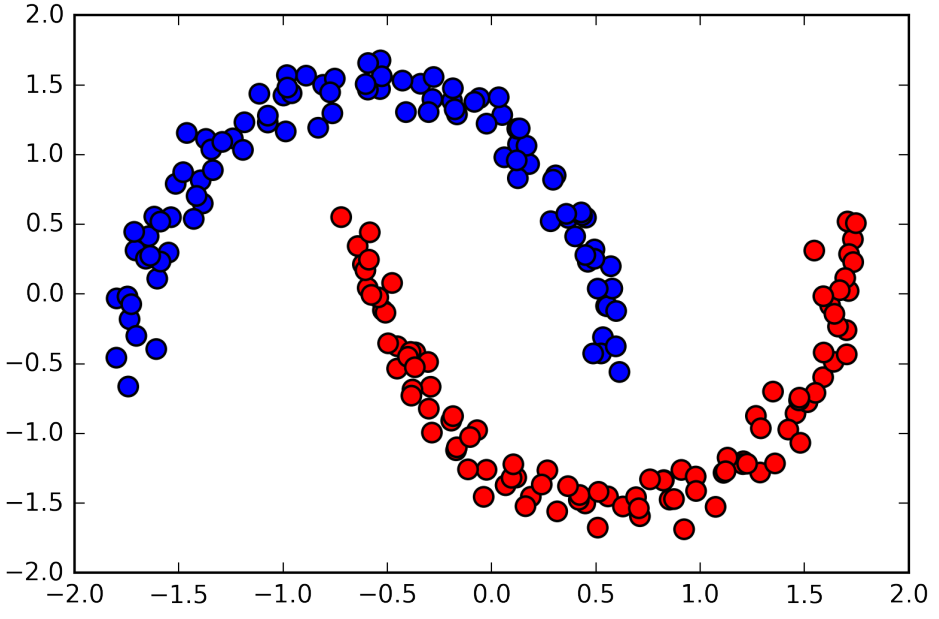
While DBSCAN doesn't require setting the number of clusters explicitly, setting `eps` implicitly controls how many clusters will be found.

Finding a good setting for `eps` is sometimes easier after scaling the data using `StandardScaler` or `MinMaxScaler`, as using these scaling techniques will ensure that all features have similar ranges.

Below is the result of DBSCAN running on the `two_moons` dataset. The algorithm actually finds the two half-circles and separates them using the default settings.

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# Rescale the data to zero mean and unit variance
```

```
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

dbscan = DBSCAN()
clusters = dbscan.fit_predict(X_scaled)
# plot the cluster assignments
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mglearn.cm2, s=60)
```



As the algorithm produced the desired number of clusters (two), the parameter settings seem to work well.

If we decrease `eps` to 0.2 (from the default of 0.5), we will get 8 clusters, which are clearly too many. Increasing `eps` to 0.7 results in a single cluster.

When using DBSCAN, you need to be careful about handling the returned cluster assignments. The use of `-1` to indicate noise might result in unexpected effects when using the cluster labels to index another array. [an example here could be useful - sort of know what you're talking about, but sort of not]

Comparing and evaluating clustering algorithms

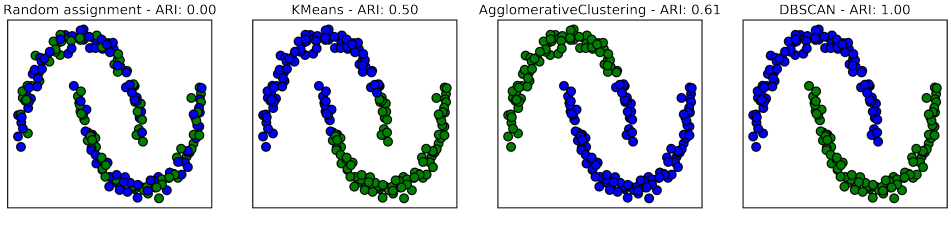
After talking about the algorithms behind k-Means, agglomerative clustering and DBSCAN, we will now compare them on some real world datasets. One of the challenges in applying clustering algorithms is that it is very hard to access how well a clustering algorithm worked, and to compare outcomes between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

Evaluating clustering with ground truth

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

Below we compare the k-Means, agglomerative clustering and DBSCAN algorithms using ARI. We also include what it looks like when we randomly assign points to two clusters for comparison.

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

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

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

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

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

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

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



The adjusted rand index provides intuitive results, with a random cluster assignment having a score of 0, and DBSCAN (which recovers the desired clustering perfectly) having a score of 1. A common mistake when evaluating clustering in this way is to

use `accuracy_score` instead of a clustering metric like `adjusted_rand_score` and `normalized_mutual_info_score`.

The problem in using accuracy is that it requires the assigned cluster labels to exactly match the ground truth. However, the cluster labels themselves are meaningless, and only which points are in the same cluster matters:

```
from sklearn.metrics import accuracy_score

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

```
Accuracy: 0.00

ARI: 1.00
```

Evaluating clustering without ground truth

Although we have just shown how to evaluate on clustering algorithms, in practice, there is a big problem with the evaluation using measures like ARI.

When applying clustering algorithms, there is usually no ground truth to which to compare. If we knew the right clustering of the data, we could use this information to build a supervised model like a classifier.

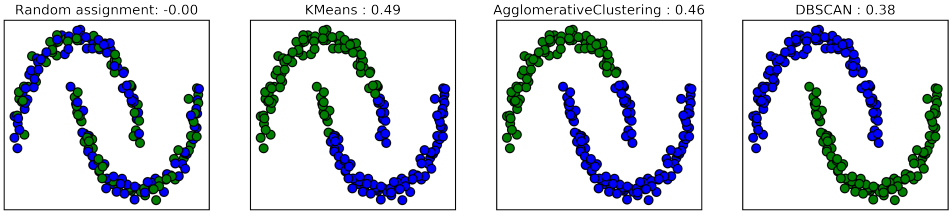
Therefore, using metric like ARI and NMI usually only really helps in developing algorithms, not in assessing success in an application.

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

Here is an example comparing the outcome of k-Means, agglomerative clustering and DBSCAN on the two moons using the silhouette score:

```
from sklearn.metrics.cluster import silhouette_score

X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

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

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

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

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

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

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



As you can see, k-Means gets the highest silhouette score, even though we might prefer the result produced by DBSCAN.

A slightly better strategy for evaluating clusters are *robustness-based* clustering metrics. These run an algorithm after adding some noise to the data, or using different parameter settings, and compare the outcomes. The idea is that if many algorithm parameters and many perturbations of the data return the same result, it is likely to be trustworthy.

Unfortunately, this strategy is not implemented in scikit-learn at the time of writing.

Even if we get a very robust clustering, or a very high silhouette score, we still don't know if there is any semantic meaning in the clustering, or whether the clustering reflects an aspect of the data that we are interested in.

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

Only if we actually analyze the clusters can we know whether the clustering corresponds to anything we are interested in.

Comparing algorithms on the faces dataset

Let's apply the k-Means, DBSCAN and agglomerative clustering algorithms to the labeled faces in the wild dataset, and see if any of them find interesting structure.

We will use the eigenface representation of the data, as produced by `PCA(whiten=True)`, with 100 components. We saw above that this is a more semantic representation of the face images than the raw pixels. It will also make computation faster.

It's a good exercise for you to run the experiments below on the original data and compare results.

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

We will start by applying DBSCAN, which we just discussed.

Analyzing the faces dataset with DBSCAN

```
# apply DBSCAN with default parameters
dbscan = DBSCAN()
labels = dbscan.fit_predict(X_pca)
np.unique(labels)
```

```
array([-1])
```

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

```
dbscan = DBSCAN(min_samples=3)
labels = dbscan.fit_predict(X_pca)
np.unique(labels)
```

```
array([-1])
```

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

```
dbscan = DBSCAN(min_samples=3, eps=15)
labels = dbscan.fit_predict(X_pca)
np.unique(labels)
```

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

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

```
# count number of points in all clusters and noise.
# bincount doesn't allow negative numbers, so we need to add 1.
# the first number in the result corresponds to noise points
np.bincount(labels + 1)
```

```
array([  27, 2036])
```

There are only very few noise points, 27, so we can look at all of them:

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

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

Comparing these images to the random sample of face images from Figure some_faces, we can guess why they were labeled as noise: the image in the sixth image in the first row one has a person drinking from a glass, there are images with hats, and the second to last image has a hand in front of the face. The other images contain odd angles or crops that are too close (see the first image in the first row) or too wide (see the last image in the first row).

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

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

```
for eps in [1, 3, 5, 7, 9, 11, 13]:
    print("\neps=%d" % eps)
    dbscan = DBSCAN(eps=eps, min_samples=3)
    labels = dbscan.fit_predict(X_pca)
    print("Number of clusters: %s" % np.unique(labels))
    print("Clusters: %s" % np.bincount(labels + 1))
```

```
eps=1

Number of clusters: [-1]

Clusters: [2063]


eps=3

Number of clusters: [-1]

Clusters: [2063]


eps=5

Number of clusters: [-1]

Clusters: [2063]


eps=7

Number of clusters: [-1  0  1  2  3  4  5  6  7  8  9 10 11 12]

Clusters: [2008    4    6    3    6    9    5    3    3    4    3    3    3    3]


eps=9

Number of clusters: [-1  0  1  2]

Clusters: [1273  784    3    3]


eps=11

Number of clusters: [-1  0]
```

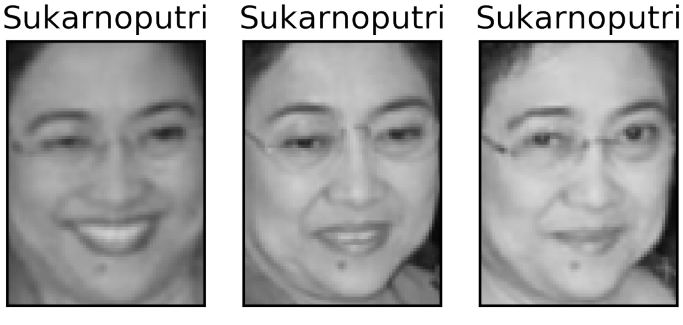
```
Clusters: [ 429 1634]


eps=13

Number of clusters: [-1  0]

Clusters: [ 115 1948]
```

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

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

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

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

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

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



Some of the clusters correspond to people with very distinct faces (within this dataset), such as Tayyip or Koizumi. Within each cluster, the orientation of the face is also quite fixed, as well as the facial expression. Some of the clusters contain faces of multiple people, but they share a similar orientation of the face and expression.

This concludes our analysis of the DBSCAN algorithm applied to the faces dataset.

As you can see, we are doing a very manual analysis here, different from the much more automatic search approach we could use for supervised learning, based on R^2 or accuracy.

Let's move on to applying k-Means and Agglomerative Clustering.

Analyzing the faces dataset with k-Means

We saw that it was not possible to create more than one big cluster using DBSCAN. Agglomerative clustering and k-Means are much more likely to create clusters of even size, but we do need to set a number of clusters.

We could set the number of clusters to the known number of people in the dataset, though it is very unlikely that an unsupervised clustering algorithm will recover them.

Instead, we can start with a low number of clusters, like 10, which might allow us to analyze each of the clusters.

```
n_clusters = 10
# extract clusters with k-Means
km = KMeans(n_clusters=n_clusters, random_state=0)
labels_km = km.fit_predict(X_pca)
print("cluster sizes k-Means: %s" % np.bincount(labels_km))
```

```
cluster sizes k-Means: [185 146 168 190 153 284 263 133 223 318]
```

As you can see, k-Means clustering partitioned the data into relatively similarly sized clusters from 133 to 318. This is quite different from the result of DBSCAN.

We can further analyze the outcome of k-Means by visualizing the cluster centers. As we clustered in the representation produced by PCA, we need to rotate the cluster centers back into the original space to visualize them, using `pca.inverse_transform`.

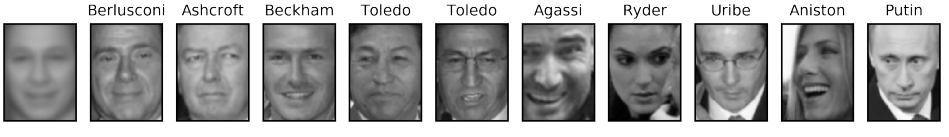
```
fig, axes = plt.subplots(2, 5, subplot_kw={'xticks': (), 'yticks': ()}, figsize=(12, 4))
for center, ax in zip(km.cluster_centers_, axes.ravel()):
    ax.imshow(pca.inverse_transform(center).reshape(image_shape), vmin=0, vmax=1)
```

The cluster centers found by k-Means are very smooth version of faces. This is not very surprising, given that each center is an average of 133 to 318 face images. Working with a reduced PCA representation adds to the smoothness of the images (compared to faces reconstructed using 100 PCA dimensions in Figure pca_reconstructions).

The clustering seems to pick up on different orientations of the face, different expression (the third cluster center seems to show a smiling face), and presence of collars (see the second to last cluster center).

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

```
n_clusters = 10
for cluster in range(n_clusters):
    center = km.cluster_centers_[cluster]
    mask = km.labels_ == cluster
    dists = np.sum((X_pca - center) ** 2, axis=1)
    dists[~mask] = np.inf
    inds = np.argsort(dists)[:5]
    dists[~mask] = -np.inf
    inds = np.r_[inds, np.argsort(dists)[-5:]]
    fig, axes = plt.subplots(1, 11, subplot_kw={'xticks': (), 'yticks': ()}, figsize=(10, 8))
    axes[0].imshow(pca.inverse_transform(center).reshape(image_shape), vmin=0, vmax=1)
    for image, label, asdf, ax in zip(X_people[inds], y_people[inds], labels_km[inds], axes[1:]):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title("%s" % (people.target_names[label].split()[-1]), fontdict={'fontsize': 9})
print("kmeans_face_clusters")
```



```
kmeans_face_clusters
```

Figure kmeans_face_clusters confirms our intuition about smiling faces for the third cluster, and also the importance of orientation for the other clusters. The "atypical" points are not very similar to the cluster center, though, and the assignment seems somewhat arbitrary. This can be attributed to the fact that k-Means partitions all the data points, and doesn't have a concept of "noise" points, as DBSCAN does.

Using a larger number of clusters, the algorithm could find finer distinctions. However, adding more clusters makes manual inspection even harder.

Analyzing the faces dataset with agglomerative clustering

Now, let's look at the results of agglomerative clustering:

```
# extract clusters with ward agglomerative clustering
agglomerative = AgglomerativeClustering(n_clusters=10)
labels_agg = agglomerative.fit_predict(X_pca)
print("cluster sizes agglomerative clustering: %s" % np.bincount(labels_agg))
```

```
cluster sizes agglomerative clustering: [167  50 252 367 160  80  50  67 521 349]
```

Agglomerative clustering produces relatively equally sized clusters, with cluster sizes between 50 and 521. These are more uneven than k-Means, but much more even than the ones produced by DBSCAN.

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

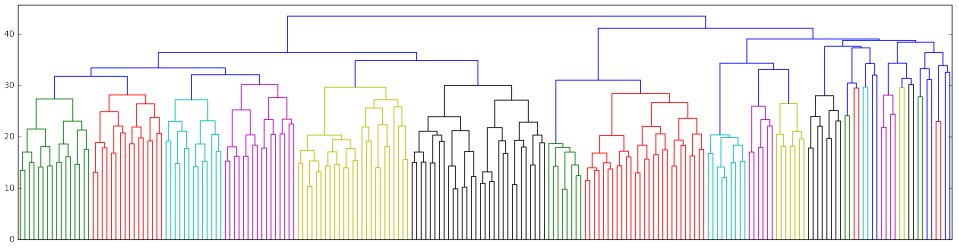
```
adjusted_rand_score(labels_agg, labels_km)
```

```
0.09733057984938646
```

An ARI of only 0.1 means that the two clusterings `labels_agg` and `labels_km` have quite little in common. This is not very surprising, given the fact that points further away from the cluster centers seem to have little in common for k-Means.

Next, we might want to plot the dendrogram. We limit the depth of the tree in the plot, as branching down to the individual 2063 data points would result in an unreadably dense plot.

```
# import the dendogram function and the ward clustering function from scipy
from scipy.cluster.hierarchy import dendrogram, ward

# apply the ward clustering to the data array X
# The scipy ward function returns an array that specifies the distances bridged
# when performing agglomerative clustering
linkage_array = ward(X_pca)
# now we plot the dendogram for the linkage_array containing the distances between clusters
plt.figure(figsize=(20, 5))
dendrogram(linkage_array, p=7, truncate_mode='level', no_labels=True);
```



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

Here is a visualization of the ten clusters, similarly to k-Means above.

Note that there is no notion of cluster center in agglomerative clustering (though we could compute the mean), and we simply show the first couple of points in each cluster. We show the number of points in each cluster to the left of the first image.

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

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

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

n_clusters = 40
for cluster in [15, 7, 17, 20, 25, 29]: # hand-picked "interesting" clusters
```

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


```
cluster sizes agglomerative clustering: [ 50 111 103 120  60 169  50  53  10  33 187  30  50  85

 111  18   3  78  81  36  10  16  22   5  57  27  75  26  92   5  13  35

   8  23  20  69]
```

Here, the clustering seems to have picked up on "dark skinned and smiling", "serious eyebrows", "collared shirt", "white hat and showing front teeth", "high forehead" and "Asian".

We could also] find these highly similar clusters using the dendrogram, if we did more a detailed analysis.

Summary of Clustering Methods

We saw above that applying and evaluating clustering is a highly qualitative procedure, and often most helpful in the exploratory phase of data analysis. We looked at three clustering algorithms, k-Means, DBSCAN and Agglomerative Clustering. All three have a way of controlling the granularity of clustering. k-Means and Agglomerative Clustering allow you to directly specify the number of desired clusters, while DBSCAN lets you define proximity using the `eps` parameter, which indirectly influences cluster size.

All three methods can be used on large, real-world datasets, are relatively easy to understand, and allow for clustering into many clusters.

Each of the algorithms has somewhat different strengths. k-Means allows for a characterization of the clusters using the cluster means. It can also be viewed as a decomposition method, where each data point is represented by its cluster center.

DBSCAN allows for the detection of "noise points" that are not assigned any cluster, and it can help automatically determine the number of clusters. In contrast to the other two methods, it allow for complex cluster shapes, as we saw in the `two_moons` example. DBSCAN sometimes produces clusters of very differing size, which can be a

strength or a weakness. Agglomerative clustering can provide a whole hierarchy of possible partitions of the data, which can be easily inspected via dendrograms.

Summary and Outlook

This chapter introduced a range of unsupervised learning algorithms that can be applied for exploratory data analysis and preprocessing. Having the right representation of the data is often crucial for supervised or unsupervised learning to succeed, and preprocessing and decomposition methods play an important part in data preparation.

Decomposition, manifold learning and clustering are essential tools to further your understanding of your data, and can be the only way to make sense of your data in the absence of supervision information. Even in the supervised setting, exploratory tools are important for a better understanding of the properties of the data.

Often it is hard to quantify the usefulness of an unsupervised algorithm, though this shouldn't deter you from using them to create insights from your data.

With these methods under your belt, you are now equipped with all the essential learning algorithms that machine learning practitioners use every day.

We encourage you to try clustering and decomposition methods both on two-dimensional toy data, as well as real world datasets included in scikit-learn, like the `digits`, `iris` and `cancer` data sets.

CHAPTER 4

Summary of scikit-learn methods and usage

In this chapter, we briefly recapitulate the main parts of the scikit-learn API that we have seen so far, as well as show some ways to simplify your code.

The Estimator Interface

All algorithms in scikit-learn, whether preprocessing, supervised learning or unsupervised learning algorithms are all implemented as classes. These classes are called *estimators* in scikit-learn. To apply an algorithm, you first have to instantiate an object of the particular class:

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
```

The estimator class contains the algorithm, and also stored the model that is learned from data using the algorithm.

When constructing the model object, this is also the time when you should set any parameters of the model. These parameters include regularization, complexity control, number of clusters to find, etc, as we discussed in detail in Chapter 2 and Chapter 3.

All estimators have a `fit` method, which is used to build the model. The `fit` method always requires as first argument the data `X`, represented as a numpy array or a scipy sparse matrix, where each row represents a single data point. The data `X` is always assumed to be a numpy array or scipy sparse matrix that has continuous (floating point) entries.

Supervised algorithms also require a `y` argument, which is a one-dimensional numpy array, containing target values for regression or classification, i.e. the known output labels or responses.

There are two main ways to apply a learned model in scikit-learn. To create a prediction in the form of a new output like `y`, you use the `predict` method. To create a new representation of the input data `X`, you use the `transform` method. Table api_summary summarizes the use-cases of the predict and transform methods.

`estimator.fit(X_train, [y_train])`	
`estimator.predict(X_test)`	**`estimator.transform(X_test)`**
Classification	Preprocessing
Regression	Dimensionality Reduction
Clustering	Feature Extraction
	Feature selection

Table api_summary

Additionally, all supervised models have a `score(X_test, y_test)` method, that allows an evaluation of the model.

Here `X_train` and `y_train` refere to the training data and training labels, while `X_test` and `y_test` refer to the test data and test labels (if applicable).

Fit resets a model

An important property of scikit-learn models is that calling `fit` will always reset everything a model previously learned. So if you build a model on one dataset, and then call `fit` again on a different dataset, the model will "forget" everything it learned from the first data. You can call `fit` as often as you like on a model, and the outcome will be the same as calling `fit` on a "new" model:

```
# get some data
from sklearn.datasets import make_blobs, load_iris
from sklearn.model_selection import train_test_split

# load iris
iris = load_iris()

# create some blobs
X, y = make_blobs(random_state=0, centers=4)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# build a model on the iris dataset
logreg = LogisticRegression()
```