



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Building Machine Learning Systems with Python

Master the art of machine learning with Python and build effective machine learning systems with this intensive hands-on guide

Willi Richert
Luis Pedro Coelho

[PACKT] open source*
PUBLISHING community experience distilled

Table of Contents

Preface	1
Chapter 1: Getting Started with Python Machine Learning	7
Machine learning and Python – the dream team	8
What the book will teach you (and what it will not)	9
What to do when you are stuck	10
Getting started	11
Introduction to NumPy, SciPy, and Matplotlib	12
Installing Python	12
Chewing data efficiently with NumPy and intelligently with SciPy	12
Learning NumPy	13
Indexing	15
Handling non-existing values	15
Comparing runtime behaviors	16
Learning SciPy	17
Our first (tiny) machine learning application	19
Reading in the data	19
Preprocessing and cleaning the data	20
Choosing the right model and learning algorithm	22
Before building our first model	22
Starting with a simple straight line	22
Towards some advanced stuff	24
Stepping back to go forward – another look at our data	26
Training and testing	28
Answering our initial question	30
Summary	31
Chapter 2: Learning How to Classify with Real-world Examples	33
The Iris dataset	33
The first step is visualization	34
Building our first classification model	35
Evaluation – holding out data and cross-validation	38

Building more complex classifiers	40
A more complex dataset and a more complex classifier	41
Learning about the Seeds dataset	42
Features and feature engineering	43
Nearest neighbor classification	44
Binary and multiclass classification	47
Summary	48
Chapter 3: Clustering – Finding Related Posts	49
Measuring the relatedness of posts	50
How not to do it	50
How to do it	51
Preprocessing – similarity measured as similar number of common words	51
Converting raw text into a bag-of-words	52
Counting words	53
Normalizing the word count vectors	56
Removing less important words	56
Stemming	57
Installing and using NLTK	58
Extending the vectorizer with NLTK's stemmer	59
Stop words on steroids	60
Our achievements and goals	61
Clustering	62
KMeans	63
Getting test data to evaluate our ideas on	65
Clustering posts	67
Solving our initial challenge	68
Another look at noise	71
Tweaking the parameters	72
Summary	73
Chapter 4: Topic Modeling	75
Latent Dirichlet allocation (LDA)	75
Building a topic model	76
Comparing similarity in topic space	80
Modeling the whole of Wikipedia	83
Choosing the number of topics	86
Summary	87
Chapter 5: Classification – Detecting Poor Answers	89
Sketching our roadmap	90
Learning to classify classy answers	90

Tuning the instance	90
Tuning the classifier	90
Fetching the data	91
Slimming the data down to chewable chunks	92
Preselection and processing of attributes	93
Defining what is a good answer	94
Creating our first classifier	95
Starting with the k-nearest neighbor (kNN) algorithm	95
Engineering the features	96
Training the classifier	97
Measuring the classifier's performance	97
Designing more features	98
Deciding how to improve	101
Bias-variance and its trade-off	102
Fixing high bias	102
Fixing high variance	103
High bias or low bias	103
Using logistic regression	105
A bit of math with a small example	106
Applying logistic regression to our postclassification problem	108
Looking behind accuracy – precision and recall	110
Slimming the classifier	114
Ship it!	115
Summary	115
Chapter 6: Classification II – Sentiment Analysis	117
Sketching our roadmap	117
Fetching the Twitter data	118
Introducing the Naive Bayes classifier	118
Getting to know the Bayes theorem	119
Being naive	120
Using Naive Bayes to classify	121
Accounting for unseen words and other oddities	124
Accounting for arithmetic underflows	125
Creating our first classifier and tuning it	127
Solving an easy problem first	128
Using all the classes	130
Tuning the classifier's parameters	132
Cleaning tweets	136
Taking the word types into account	138
Determining the word types	139

Successfully cheating using SentiWordNet	141
Our first estimator	143
Putting everything together	145
Summary	146
Chapter 7: Regression – Recommendations	147
Predicting house prices with regression	147
Multidimensional regression	151
Cross-validation for regression	151
Penalized regression	153
L1 and L2 penalties	153
Using Lasso or Elastic nets in scikit-learn	154
P greater than N scenarios	155
An example based on text	156
Setting hyperparameters in a smart way	158
Rating prediction and recommendations	159
Summary	163
Chapter 8: Regression – Recommendations Improved	165
Improved recommendations	165
Using the binary matrix of recommendations	166
Looking at the movie neighbors	168
Combining multiple methods	169
Basket analysis	172
Obtaining useful predictions	173
Analyzing supermarket shopping baskets	173
Association rule mining	176
More advanced basket analysis	178
Summary	179
Chapter 9: Classification III – Music Genre Classification	181
Sketching our roadmap	181
Fetching the music data	182
Converting into a wave format	182
Looking at music	182
Decomposing music into sine wave components	184
Using FFT to build our first classifier	186
Increasing experimentation agility	186
Training the classifier	187
Using the confusion matrix to measure accuracy in multiclass problems	188
An alternate way to measure classifier performance using receiver operator characteristic (ROC)	190

Improving classification performance with Mel Frequency Cepstral	
Coefficients	193
Summary	197
Chapter 10: Computer Vision – Pattern Recognition	199
<hr/>	
Introducing image processing	199
Loading and displaying images	200
Basic image processing	201
Thresholding	202
Gaussian blurring	205
Filtering for different effects	207
Adding salt and pepper noise	207
Putting the center in focus	208
Pattern recognition	210
Computing features from images	211
Writing your own features	212
Classifying a harder dataset	215
Local feature representations	216
Summary	219
Chapter 11: Dimensionality Reduction	221
<hr/>	
Sketching our roadmap	222
Selecting features	222
Detecting redundant features using filters	223
Correlation	223
Mutual information	225
Asking the model about the features using wrappers	230
Other feature selection methods	232
Feature extraction	233
About principal component analysis (PCA)	233
Sketching PCA	234
Applying PCA	234
Limitations of PCA and how LDA can help	236
Multidimensional scaling (MDS)	237
Summary	240
Chapter 12: Big(ger) Data	241
<hr/>	
Learning about big data	241
Using jug to break up your pipeline into tasks	242
About tasks	242
Reusing partial results	245
Looking under the hood	246
Using jug for data analysis	246

Table of Contents

Using Amazon Web Services (AWS)	248
Creating your first machines	250
Installing Python packages on Amazon Linux	253
Running jug on our cloud machine	254
Automating the generation of clusters with starcluster	255
Summary	259
Appendix: Where to Learn More about Machine Learning	261
Online courses	261
Books	261
Q&A sites	262
Blogs	262
Data sources	263
Getting competitive	263
What was left out	264
Summary	264
Index	265

Preface

You could argue that it is a fortunate coincidence that you are holding this book in your hands (or your e-book reader). After all, there are millions of books printed every year, which are read by millions of readers; and then there is this book read by you. You could also argue that a couple of machine learning algorithms played their role in leading you to this book (or this book to you). And we, the authors, are happy that you want to understand more about the how and why.

Most of this book will cover the how. How should the data be processed so that machine learning algorithms can make the most out of it? How should you choose the right algorithm for a problem at hand?

Occasionally, we will also cover the why. Why is it important to measure correctly? Why does one algorithm outperform another one in a given scenario?

We know that there is much more to learn to be an expert in the field. After all, we only covered some of the "hows" and just a tiny fraction of the "whys". But at the end, we hope that this mixture will help you to get up and running as quickly as possible.

What this book covers

Chapter 1, Getting Started with Python Machine Learning, introduces the basic idea of machine learning with a very simple example. Despite its simplicity, it will challenge us with the risk of overfitting.

Chapter 2, Learning How to Classify with Real-world Examples, explains the use of real data to learn about classification, whereby we train a computer to be able to distinguish between different classes of flowers.

Chapter 3, Clustering – Finding Related Posts, explains how powerful the bag-of-words approach is when we apply it to finding similar posts without really understanding them.

Chapter 4, Topic Modeling, takes us beyond assigning each post to a single cluster and shows us how assigning them to several topics as real text can deal with multiple topics.

Chapter 5, Classification – Detecting Poor Answers, explains how to use logistic regression to find whether a user's answer to a question is good or bad. Behind the scenes, we will learn how to use the bias-variance trade-off to debug machine learning models.

Chapter 6, Classification II – Sentiment Analysis, introduces how Naive Bayes works, and how to use it to classify tweets in order to see whether they are positive or negative.

Chapter 7, Regression – Recommendations, discusses a classical topic in handling data, but it is still relevant today. We will use it to build recommendation systems, a system that can take user input about the likes and dislikes to recommend new products.

Chapter 8, Regression – Recommendations Improved, improves our recommendations by using multiple methods at once. We will also see how to build recommendations just from shopping data without the need of rating data (which users do not always provide).

Chapter 9, Classification III – Music Genre Classification, illustrates how if someone has scrambled our huge music collection, then our only hope to create an order is to let a machine learner classify our songs. It will turn out that it is sometimes better to trust someone else's expertise than creating features ourselves.

Chapter 10, Computer Vision – Pattern Recognition, explains how to apply classifications in the specific context of handling images, a field known as pattern recognition.

Chapter 11, Dimensionality Reduction, teaches us what other methods exist that can help us in downsizing data so that it is chewable by our machine learning algorithms.

Chapter 12, Big(ger) Data, explains how data sizes keep getting bigger, and how this often becomes a problem for the analysis. In this chapter, we explore some approaches to deal with larger data by taking advantage of multiple core or computing clusters. We also have an introduction to using cloud computing (using Amazon's Web Services as our cloud provider).

Appendix, Where to Learn More about Machine Learning, covers a list of wonderful resources available for machine learning.

What you need for this book

This book assumes you know Python and how to install a library using `easy_install` or `pip`. We do not rely on any advanced mathematics such as calculus or matrix algebra.

To summarize it, we are using the following versions throughout this book, but you should be fine with any more recent one:

- Python: 2.7
- NumPy: 1.6.2
- SciPy: 0.11
- Scikit-learn: 0.13

Who this book is for

This book is for Python programmers who want to learn how to perform machine learning using open source libraries. We will walk through the basic modes of machine learning based on realistic examples.

This book is also for machine learners who want to start using Python to build their systems. Python is a flexible language for rapid prototyping, while the underlying algorithms are all written in optimized C or C++. Therefore, the resulting code is fast and robust enough to be usable in production as well.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive".


A block of code is set as follows:


```
def nn_movie(movie_likeness, reviews, uid, mid):
    likes = movie_likeness[mid].argsort()
    # reverse the sorting so that most alike are in
    # beginning
    likes = likes[::-1]
    # returns the rating for the most similar movie available
    for e11 in likes:
        if reviews[u,e11] > 0:
            return reviews[u,e11]
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
def nn_movie(movie_likeness, reviews, uid, mid):
    likes = movie_likeness[mid].argsort()
    # reverse the sorting so that most alike are in
    # beginning
    likes = likes[::-1]
    # returns the rating for the most similar movie available
    for ell in likes:
        if reviews[u,ell] > 0:
            return reviews[u,ell]
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking on the **Next** button moves you to the next screen".

 [Warnings or important notes appear in a box like this.]

 [Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Getting Started with Python Machine Learning

Machine learning (ML) teaches machines how to carry out tasks by themselves. It is that simple. The complexity comes with the details, and that is most likely the reason you are reading this book.

Maybe you have too much data and too little insight, and you hoped that using machine learning algorithms will help you solve this challenge. So you started to dig into random algorithms. But after some time you were puzzled: which of the myriad of algorithms should you actually choose?

Or maybe you are broadly interested in machine learning and have been reading a few blogs and articles about it for some time. Everything seemed to be magic and cool, so you started your exploration and fed some toy data into a decision tree or a support vector machine. But after you successfully applied it to some other data, you wondered, was the whole setting right? Did you get the optimal results? And how do you know there are no better algorithms? Or whether your data was "the right one"?

Welcome to the club! We, the authors, were at those stages once upon a time, looking for information that tells the real story behind the theoretical textbooks on machine learning. It turned out that much of that information was "black art", not usually taught in standard textbooks. So, in a sense, we wrote this book to our younger selves; a book that not only gives a quick introduction to machine learning, but also teaches you lessons that we have learned along the way. We hope that it will also give you, the reader, a smoother entry into one of the most exciting fields in Computer Science.

Machine learning and Python – the dream team

The goal of machine learning is to teach machines (software) to carry out tasks by providing them with a couple of examples (how to do or not do a task). Let us assume that each morning when you turn on your computer, you perform the same task of moving e-mails around so that only those e-mails belonging to a particular topic end up in the same folder. After some time, you feel bored and think of automating this chore. One way would be to start analyzing your brain and writing down all the rules your brain processes while you are shuffling your e-mails. However, this will be quite cumbersome and always imperfect. While you will miss some rules, you will over-specify others. A better and more future-proof way would be to automate this process by choosing a set of e-mail meta information and body/folder name pairs and let an algorithm come up with the best rule set. The pairs would be your training data, and the resulting rule set (also called model) could then be applied to future e-mails that we have not yet seen. This is machine learning in its simplest form.

Of course, machine learning (often also referred to as data mining or predictive analysis) is not a brand new field in itself. Quite the contrary, its success over recent years can be attributed to the pragmatic way of using rock-solid techniques and insights from other successful fields; for example, statistics. There, the purpose is for us humans to get insights into the data by learning more about the underlying patterns and relationships. As you read more and more about successful applications of machine learning (you have checked out kaggle.com already, haven't you?), you will see that applied statistics is a common field among machine learning experts.

As you will see later, the process of coming up with a decent ML approach is never a waterfall-like process. Instead, you will see yourself going back and forth in your analysis, trying out different versions of your input data on diverse sets of ML algorithms. It is this explorative nature that lends itself perfectly to Python. Being an interpreted high-level programming language, it may seem that Python was designed specifically for the process of trying out different things. What is more, it does this very fast. Sure enough, it is slower than C or similar statically-typed programming languages; nevertheless, with a myriad of easy-to-use libraries that are often written in C, you don't have to sacrifice speed for agility.

What the book will teach you (and what it will not)

This book will give you a broad overview of the types of learning algorithms that are currently used in the diverse fields of machine learning and what to watch out for when applying them. From our own experience, however, we know that doing the "cool" stuff – using and tweaking machine learning algorithms such as **support vector machines (SVM)**, **nearest neighbor search (NNS)**, or ensembles thereof – will only consume a tiny fraction of the overall time of a good machine learning expert. Looking at the following typical workflow, we see that most of our time will be spent in rather mundane tasks:

1. Reading the data and cleaning it.
2. Exploring and understanding the input data.
3. Analyzing how best to present the data to the learning algorithm.
4. Choosing the right model and learning algorithm.
5. Measuring the performance correctly.

When talking about exploring and understanding the input data, we will need a bit of statistics and basic math. But while doing this, you will see that those topics, which seemed so dry in your math class, can actually be really exciting when you use them to look at interesting data.

The journey begins when you read in the data. When you have to face issues such as invalid or missing values, you will see that this is more an art than a precise science. And a very rewarding one, as doing this part right will open your data to more machine learning algorithms, and thus increase the likelihood of success.

With the data being ready in your program's data structures, you will want to get a real feeling of what kind of animal you are working with. Do you have enough data to answer your questions? If not, you might want to think about additional ways to get more of it. Do you maybe even have too much data? Then you probably want to think about how best to extract a sample of it.

Often you will not feed the data directly into your machine learning algorithm. Instead, you will find that you can refine parts of the data before training. Many times, the machine learning algorithm will reward you with increased performance. You will even find that a simple algorithm with refined data generally outperforms a very sophisticated algorithm with raw data. This part of the machine learning workflow is called **feature engineering**, and it is generally a very exciting and rewarding challenge. Creative and intelligent that you are, you will immediately see the results.

Choosing the right learning algorithm is not simply a shootout of the three or four that are in your toolbox (there will be more algorithms in your toolbox that you will see). It is more of a thoughtful process of weighing different performance and functional requirements. Do you need fast results and are willing to sacrifice quality? Or would you rather spend more time to get the best possible result? Do you have a clear idea of the future data or should you be a bit more conservative on that side?

Finally, measuring the performance is the part where most mistakes are waiting for the aspiring ML learner. There are easy ones, such as testing your approach with the same data on which you have trained. But there are more difficult ones; for example, when you have imbalanced training data. Again, data is the part that determines whether your undertaking will fail or succeed.

We see that only the fourth point is dealing with the fancy algorithms. Nevertheless, we hope that this book will convince you that the other four tasks are not simply chores, but can be equally important if not more exciting. Our hope is that by the end of the book you will have truly fallen in love with data instead of learned algorithms.

To that end, we will not overwhelm you with the theoretical aspects of the diverse ML algorithms, as there are already excellent books in that area (you will find pointers in *Appendix, Where to Learn More about Machine Learning*). Instead, we will try to provide an intuition of the underlying approaches in the individual chapters—just enough for you to get the idea and be able to undertake your first steps. Hence, this book is by no means "the definitive guide" to machine learning. It is more a kind of starter kit. We hope that it ignites your curiosity enough to keep you eager in trying to learn more and more about this interesting field.

In the rest of this chapter, we will set up and get to know the basic Python libraries, NumPy and SciPy, and then train our first machine learning using scikit-learn. During this endeavor, we will introduce basic ML concepts that will later be used throughout the book. The rest of the chapters will then go into more detail through the five steps described earlier, highlighting different aspects of machine learning in Python using diverse application scenarios.

What to do when you are stuck

We try to convey every idea necessary to reproduce the steps throughout this book. Nevertheless, there will be situations when you might get stuck. The reasons might range from simple typos over odd combinations of package versions to problems in understanding.

In such a situation, there are many different ways to get help. Most likely, your problem will already have been raised and solved in the following excellent Q&A sites:

- <http://metaoptimize.com/ga> - This Q&A site is laser-focused on machine learning topics. For almost every question, it contains above-average answers from machine learning experts. Even if you don't have any questions, it is a good habit to check it out every now and then and read through some of the questions and answers.
- <http://stats.stackexchange.com> - This Q&A site, named Cross Validated, is similar to MetaOptimized, but focuses more on statistics problems.
- <http://stackoverflow.com> - This Q&A site is similar to the previous ones, but with a broader focus on general programming topics. It contains, for example, more questions on some of the packages that we will use in this book (SciPy and Matplotlib).
- #machinelearning on Freenode - This IRC channel is focused on machine learning topics. It is a small but very active and helpful community of machine learning experts.
- <http://www.TwoToReal.com> - This is an instant Q&A site written by us, the authors, to support you in topics that don't fit in any of the above buckets. If you post your question, we will get an instant message; if any of us are online, we will be drawn into a chat with you.

As stated at the beginning, this book tries to help you get started quickly on your machine learning journey. We therefore highly encourage you to build up your own list of machine learning-related blogs and check them out regularly. This is the best way to get to know what works and what does not.

The only blog we want to highlight right here is <http://blog.kaggle.com>, the blog of the Kaggle company, which is carrying out machine learning competitions (more links are provided in *Appendix, Where to Learn More about Machine Learning*). Typically, they encourage the winners of the competitions to write down how they approached the competition, what strategies did not work, and how they arrived at the winning strategy. If you don't read anything else, fine; but this is a must.

Getting started

Assuming that you have already installed Python (everything at least as recent as 2.7 should be fine), we need to install NumPy and SciPy for numerical operations as well as Matplotlib for visualization.

Introduction to NumPy, SciPy, and Matplotlib

Before we can talk about concrete machine learning algorithms, we have to talk about how best to store the data we will chew through. This is important as the most advanced learning algorithm will not be of any help to us if they will never finish. This may be simply because accessing the data is too slow. Or maybe its representation forces the operating system to swap all day. Add to this that Python is an interpreted language (a highly optimized one, though) that is slow for many numerically heavy algorithms compared to C or Fortran. So we might ask why on earth so many scientists and companies are betting their fortune on Python even in the highly computation-intensive areas?

The answer is that in Python, it is very easy to offload number-crunching tasks to the lower layer in the form of a C or Fortran extension. That is exactly what NumPy and SciPy do (<http://scipy.org/install.html>). In this tandem, NumPy provides the support of highly optimized multidimensional arrays, which are the basic data structure of most state-of-the-art algorithms. SciPy uses those arrays to provide a set of fast numerical recipes. Finally, Matplotlib (<http://matplotlib.org/>) is probably the most convenient and feature-rich library to plot high-quality graphs using Python.

Installing Python

Luckily, for all the major operating systems, namely Windows, Mac, and Linux, there are targeted installers for NumPy, SciPy, and Matplotlib. If you are unsure about the installation process, you might want to install Enthought Python Distribution (https://www.enthought.com/products/epd_free.php) or Python(x,y) (<http://code.google.com/p/pythonxy/wiki/Downloads>), which come with all the earlier mentioned packages included.

Chewing data efficiently with NumPy and intelligently with SciPy

Let us quickly walk through some basic NumPy examples and then take a look at what SciPy provides on top of it. On the way, we will get our feet wet with plotting using the marvelous Matplotlib package.

You will find more interesting examples of what NumPy can offer at http://www.scipy.org/Tentative_NumPy_Tutorial.

You will also find the book *NumPy Beginner's Guide - Second Edition*, Ivan Idris, Packt Publishing very valuable. Additional tutorial style guides are at <http://scipy-lectures.github.com>; you may also visit the official SciPy tutorial at <http://docs.scipy.org/doc/scipy/reference/tutorial>.

In this book, we will use NumPy Version 1.6.2 and SciPy Version 0.11.0.

Learning NumPy

So let us import NumPy and play a bit with it. For that, we need to start the Python interactive shell.

```
>>> import numpy
>>> numpy.version.full_version
1.6.2
```

As we do not want to pollute our namespace, we certainly should not do the following:

```
>>> from numpy import *
```

The `numpy.array` array will potentially shadow the `array` package that is included in standard Python. Instead, we will use the following convenient shortcut:

```
>>> import numpy as np
>>> a = np.array([0,1,2,3,4,5])
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.ndim
1
>>> a.shape
(6,)
```

We just created an array in a similar way to how we would create a list in Python. However, NumPy arrays have additional information about the shape. In this case, it is a one-dimensional array of five elements. No surprises so far.

We can now transform this array in to a 2D matrix.

```
>>> b = a.reshape((3,2))
>>> b
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> b.ndim
2
>>> b.shape
(3, 2)
```

The funny thing starts when we realize just how much the NumPy package is optimized. For example, it avoids copies wherever possible.

```
>>> b[1][0]=77
>>> b
array([[ 0,  1],
       [77,  3],
       [ 4,  5]])
>>> a
array([ 0,  1, 77,  3,  4,  5])
```

In this case, we have modified the value 2 to 77 in `b`, and we can immediately see the same change reflected in `a` as well. Keep that in mind whenever you need a true copy.

```
>>> c = a.reshape((3,2)).copy()
>>> c
array([[ 0,  1],
       [77,  3],
       [ 4,  5]])
>>> c[0][0] = -99
>>> a
array([ 0,  1, 77,  3,  4,  5])
>>> c
array([[ -99,  1],
       [ 77,  3],
       [  4,  5]])
```

Here, `c` and `a` are totally independent copies.

Another big advantage of NumPy arrays is that the operations are propagated to the individual elements.

```
>>> a*2
array([ 2,  4,  6,  8, 10])
>>> a**2
array([ 1,  4,  9, 16, 25])
Contrast that to ordinary Python lists:
>>> [1,2,3,4,5]*2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> [1,2,3,4,5]**2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

Of course, by using NumPy arrays we sacrifice the agility Python lists offer. Simple operations like adding or removing are a bit complex for NumPy arrays. Luckily, we have both at our disposal, and we will use the right one for the task at hand.

Indexing

Part of the power of NumPy comes from the versatile ways in which its arrays can be accessed.

In addition to normal list indexing, it allows us to use arrays themselves as indices.

```
>>> a[np.array([2,3,4])]
array([77,  3,  4])
```

In addition to the fact that conditions are now propagated to the individual elements, we gain a very convenient way to access our data.

```
>>> a>4
array([False, False,  True, False, False,  True], dtype=bool)
>>> a[a>4]
array([77,  5])
```

This can also be used to trim outliers.

```
>>> a[a>4] = 4
>>> a
array([0, 1, 4, 3, 4, 4])
```

As this is a frequent use case, there is a special clip function for it, clipping the values at both ends of an interval with one function call as follows:

```
>>> a.clip(0,4)
array([0, 1, 4, 3, 4, 4])
```

Handling non-existing values

The power of NumPy's indexing capabilities comes in handy when preprocessing data that we have just read in from a text file. It will most likely contain invalid values, which we will mark as not being a real number using `numpy.NAN` as follows:

```
c = np.array([1, 2, np.NAN, 3, 4]) # let's pretend we have read this
from a text file
>>> c
array([ 1.,  2., nan,  3.,  4.])
>>> np.isnan(c)
array([False, False,  True, False, False], dtype=bool)
```

```
>>> c[~np.isnan(c)]
array([ 1.,  2.,  3.,  4.])
>>> np.mean(c[~np.isnan(c)])
2.5
```

Comparing runtime behaviors

Let us compare the runtime behavior of NumPy with normal Python lists. In the following code, we will calculate the sum of all squared numbers of 1 to 1000 and see how much time the calculation will take. We do it 10000 times and report the total time so that our measurement is accurate enough.

```
import timeit
normal_py_sec = timeit.timeit('sum(x*x for x in xrange(1000))',
                              number=10000)
naive_np_sec = timeit.timeit('sum(na*na)',
                              setup="import numpy as np; na=np.
arange(1000)",
                              number=10000)
good_np_sec = timeit.timeit('na.dot(na)',
                              setup="import numpy as np; na=np.
arange(1000)",
                              number=10000)

print("Normal Python: %f sec"%normal_py_sec)
print("Naive NumPy: %f sec"%naive_np_sec)
print("Good NumPy: %f sec"%good_np_sec)

Normal Python: 1.157467 sec
Naive NumPy: 4.061293 sec
Good NumPy: 0.033419 sec
```

We make two interesting observations. First, just using NumPy as data storage (Naive NumPy) takes 3.5 times longer, which is surprising since we believe it must be much faster as it is written as a C extension. One reason for this is that the access of individual elements from Python itself is rather costly. Only when we are able to apply algorithms inside the optimized extension code do we get speed improvements, and tremendous ones at that: using the `dot()` function of NumPy, we are more than 25 times faster. In summary, in every algorithm we are about to implement, we should always look at how we can move loops over individual elements from Python to some of the highly optimized NumPy or SciPy extension functions.

However, the speed comes at a price. Using NumPy arrays, we no longer have the incredible flexibility of Python lists, which can hold basically anything. NumPy arrays always have only one datatype.

```
>>> a = np.array([1,2,3])
>>> a.dtype
dtype('int64')
```

If we try to use elements of different types, NumPy will do its best to coerce them to the most reasonable common datatype:

```
>>> np.array([1, "stringy"])
array(['1', 'stringy'], dtype='<S8')
>>> np.array([1, "stringy", set([1,2,3])])
array([1, stringy, set([1, 2, 3])], dtype=object)
```

Learning SciPy

On top of the efficient data structures of NumPy, SciPy offers a magnitude of algorithms working on those arrays. Whatever numerical-heavy algorithm you take from current books on numerical recipes, you will most likely find support for them in SciPy in one way or another. Whether it is matrix manipulation, linear algebra, optimization, clustering, spatial operations, or even Fast Fourier transformation, the toolbox is readily filled. Therefore, it is a good habit to always inspect the `scipy` module before you start implementing a numerical algorithm.

For convenience, the complete namespace of NumPy is also accessible via SciPy. So, from now on, we will use NumPy's machinery via the SciPy namespace. You can check this easily by comparing the function references of any base function; for example:

```
>>> import scipy, numpy
>>> scipy.version.full_version
0.11.0
>>> scipy.dot is numpy.dot
True
```

The diverse algorithms are grouped into the following toolboxes:

SciPy package	Functionality
<code>cluster</code>	Hierarchical clustering (<code>cluster.hierarchy</code>) Vector quantization / K-Means (<code>cluster.vq</code>)

SciPy package	Functionality
constants	Physical and mathematical constants
	Conversion methods
fftpack	Discrete Fourier transform algorithms
integrate	Integration routines
interpolate	Interpolation (linear, cubic, and so on)
io	Data input and output
linalg	Linear algebra routines using the optimized BLAS and LAPACK libraries
maxentropy	Functions for fitting maximum entropy models
ndimage	n-dimensional image package
odr	Orthogonal distance regression
optimize	Optimization (finding minima and roots)
signal	Signal processing
sparse	Sparse matrices
spatial	Spatial data structures and algorithms
special	Special mathematical functions such as Bessel or Jacobian
stats	Statistics toolkit

The toolboxes most interesting to our endeavor are `scipy.stats`, `scipy.interpolate`, `scipy.cluster`, and `scipy.signal`. For the sake of brevity, we will briefly explore some features of the `stats` package and leave the others to be explained when they show up in the chapters.

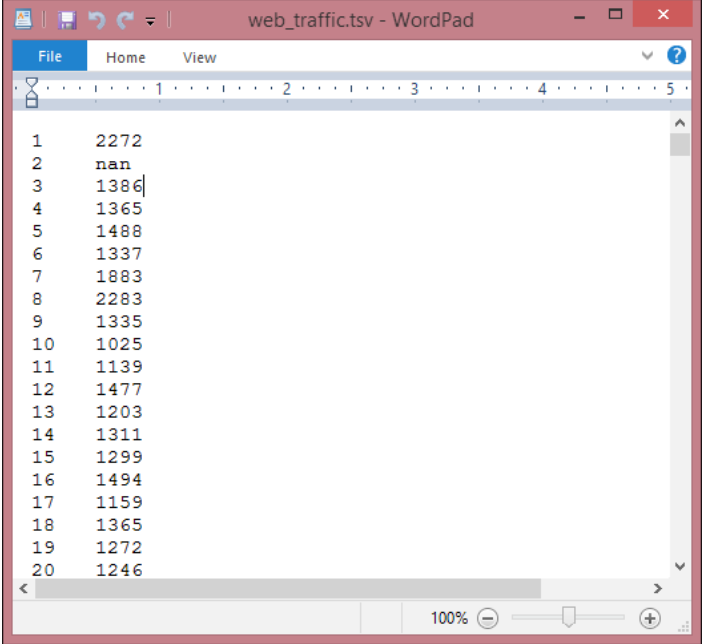
Our first (tiny) machine learning application

Let us get our hands dirty and have a look at our hypothetical web startup, MLAAS, which sells the service of providing machine learning algorithms via HTTP. With the increasing success of our company, the demand for better infrastructure also increases to serve all incoming web requests successfully. We don't want to allocate too many resources as that would be too costly. On the other hand, we will lose money if we have not reserved enough resources for serving all incoming requests. The question now is, when will we hit the limit of our current infrastructure, which we estimated being 100,000 requests per hour. We would like to know in advance when we have to request additional servers in the cloud to serve all the incoming requests successfully without paying for unused ones.

Reading in the data

We have collected the web stats for the last month and aggregated them in `ch01/data/web_traffic.tsv` (tsv because it contains tab separated values). They are stored as the number of hits per hour. Each line contains consecutive hours and the number of web hits in that hour.

The first few lines look like the following:



```
1 2272
2 nan
3 1386
4 1365
5 1488
6 1337
7 1883
8 2283
9 1335
10 1025
11 1139
12 1477
13 1203
14 1311
15 1299
16 1494
17 1159
18 1365
19 1272
20 1246
```

Using SciPy's `genfromtxt()`, we can easily read in the data.

```
import scipy as sp
data = sp.genfromtxt("web_traffic.tsv", delimiter="\t")
```

We have to specify tab as the delimiter so that the columns are correctly determined.

A quick check shows that we have correctly read in the data.

```
>>> print(data[:10])
[[ 1.00000000e+00  2.27200000e+03]
 [ 2.00000000e+00          nan]
 [ 3.00000000e+00  1.38600000e+03]
 [ 4.00000000e+00  1.36500000e+03]
 [ 5.00000000e+00  1.48800000e+03]
 [ 6.00000000e+00  1.33700000e+03]
 [ 7.00000000e+00  1.88300000e+03]
 [ 8.00000000e+00  2.28300000e+03]
 [ 9.00000000e+00  1.33500000e+03]
 [ 1.00000000e+01  1.02500000e+03]]
>>> print(data.shape)
(743, 2)
```

We have 743 data points with two dimensions.

Preprocessing and cleaning the data

It is more convenient for SciPy to separate the dimensions into two vectors, each of size 743. The first vector, x , will contain the hours and the other, y , will contain the web hits in that particular hour. This splitting is done using the special index notation of SciPy, using which we can choose the columns individually.

```
x = data[:,0]
y = data[:,1]
```



There is much more to the way data can be selected from a SciPy array. Check out http://www.scipy.org/Tentative_NumPy_Tutorial for more details on indexing, slicing, and iterating.

One caveat is that we still have some values in y that contain invalid values, `nan`. The question is, what can we do with them? Let us check how many hours contain invalid data.

```
>>> sp.sum(sp.isnan(y))
8
```

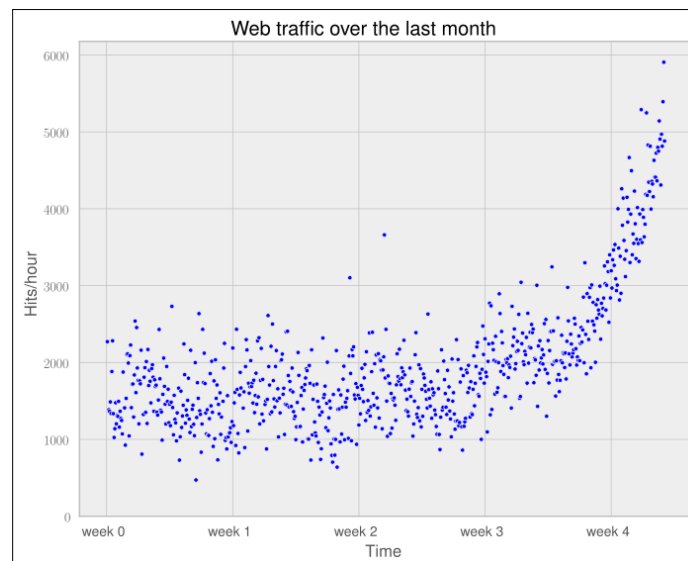
We are missing only 8 out of 743 entries, so we can afford to remove them. Remember that we can index a SciPy array with another array. `sp.isnan(y)` returns an array of Booleans indicating whether an entry is not a number. Using `~`, we logically negate that array so that we choose only those elements from `x` and `y` where `y` does contain valid numbers.

```
x = x[~sp.isnan(y)]
y = y[~sp.isnan(y)]
```

To get a first impression of our data, let us plot the data in a scatter plot using Matplotlib. Matplotlib contains the `pyplot` package, which tries to mimic Matlab's interface—a very convenient and easy-to-use one (you will find more tutorials on plotting at http://matplotlib.org/users/pyplot_tutorial.html).

```
import matplotlib.pyplot as plt
plt.scatter(x,y)
plt.title("Web traffic over the last month")
plt.xlabel("Time")
plt.ylabel("Hits/hour")
plt.xticks([w*7*24 for w in range(10)],
           ['week %i'%w for w in range(10)])
plt.autoscale(tight=True)
plt.grid()
plt.show()
```

In the resulting chart, we can see that while in the first weeks the traffic stayed more or less the same, the last week shows a steep increase:



Choosing the right model and learning algorithm

Now that we have a first impression of the data, we return to the initial question: how long will our server handle the incoming web traffic? To answer this we have to:

- Find the real model behind the noisy data points
- Use the model to extrapolate into the future to find the point in time where our infrastructure has to be extended

Before building our first model

When we talk about models, you can think of them as simplified theoretical approximations of the complex reality. As such there is always some inferiority involved, also called the approximation error. This error will guide us in choosing the right model among the myriad of choices we have. This error will be calculated as the squared distance of the model's prediction to the real data. That is, for a learned model function, f , the error is calculated as follows:

```
def error(f, x, y):
    return sp.sum((f(x)-y)**2)
```

The vectors x and y contain the web stats data that we have extracted before. It is the beauty of SciPy's vectorized functions that we exploit here with $f(x)$. The trained model is assumed to take a vector and return the results again as a vector of the same size so that we can use it to calculate the difference to y .

Starting with a simple straight line

Let us assume for a second that the underlying model is a straight line. The challenge then is how to best put that line into the chart so that it results in the smallest approximation error. SciPy's `polyfit()` function does exactly that. Given data x and y and the desired order of the polynomial (straight line has order 1), it finds the model function that minimizes the error function defined earlier.

```
fp1, residuals, rank, sv, rcond = sp.polyfit(x, y, 1, full=True)
```

The `polyfit()` function returns the parameters of the fitted model function, `fp1`; and by setting `full` to `True`, we also get additional background information on the fitting process. Of it, only residuals are of interest, which is exactly the error of the approximation.

```
>>> print("Model parameters: %s" % fp1)
Model parameters: [ 2.59619213  989.02487106]
```

```
>>> print(res)
[ 3.17389767e+08]
```

This means that the best straight line fit is the following function:

$$f(x) = 2.59619213 * x + 989.02487106.$$

We then use `poly1d()` to create a model function from the model parameters.

```
>>> f1 = sp.poly1d(fp1)
>>> print(error(f1, x, y))
317389767.34
```

We have used `full=True` to retrieve more details on the fitting process. Normally, we would not need it, in which case only the model parameters would be returned.

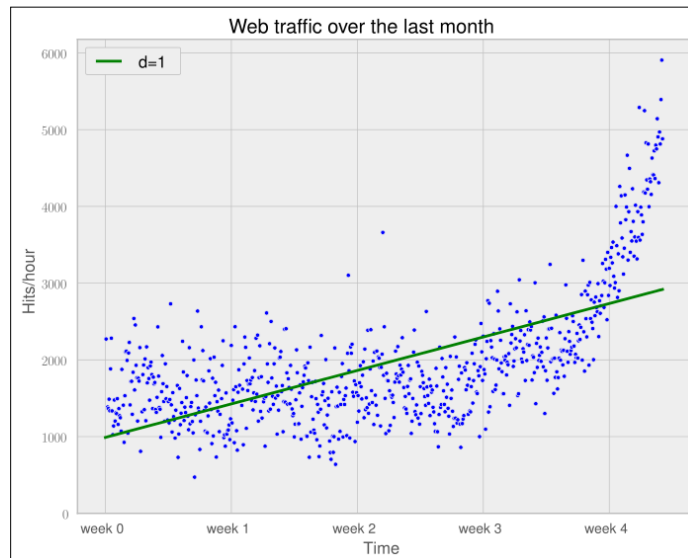


In fact, what we do here is simple curve fitting. You can find out more about it on Wikipedia by going to http://en.wikipedia.org/wiki/Curve_fitting.

We can now use `f1()` to plot our first trained model. In addition to the earlier plotting instructions, we simply add the following:

```
fx = sp.linspace(0,x[-1], 1000) # generate X-values for plotting
plt.plot(fx, f1(fx), linewidth=4)
plt.legend(["d=%i" % f1.order], loc="upper left")
```

The following graph shows our first trained model:



It seems like the first four weeks are not that far off, although we clearly see that there is something wrong with our initial assumption that the underlying model is a straight line. Plus, how good or bad actually is the error of 317,389,767.34?

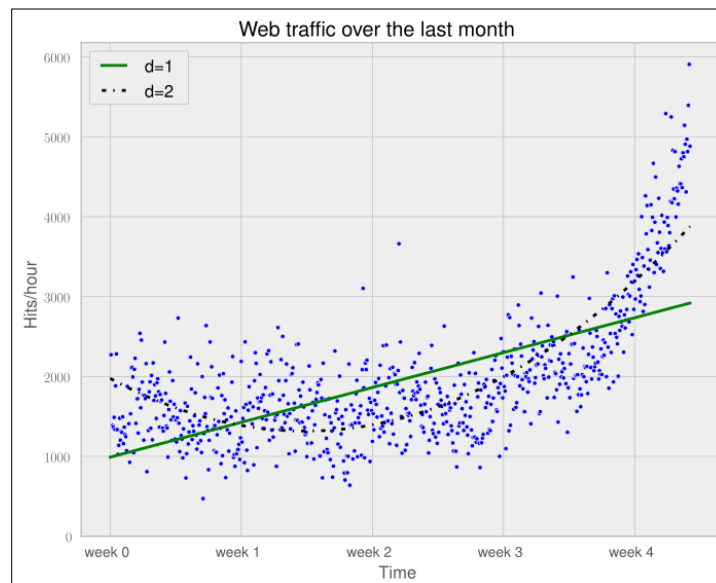
The absolute value of the error is seldom of use in isolation. However, when comparing two competing models, we can use their errors to judge which one of them is better. Although our first model clearly is not the one we would use, it serves a very important purpose in the workflow: we will use it as our baseline until we find a better one. Whatever model we will come up with in the future, we will compare it against the current baseline.

Towards some advanced stuff

Let us now fit a more complex model, a polynomial of degree 2, to see whether it better "understands" our data:

```
>>> f2p = sp.polyfit(x, y, 2)
>>> print(f2p)
array([ 1.05322215e-02, -5.26545650e+00,  1.97476082e+03])
>>> f2 = sp.poly1d(f2p)
>>> print(error(f2, x, y))
179983507.878
```

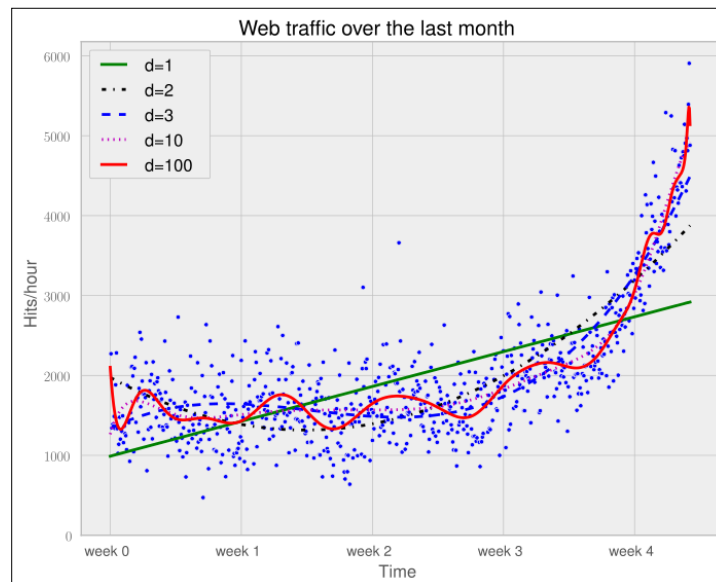
The following chart shows the model we trained before (straight line of one degree) with our newly trained, more complex model with two degrees (dashed):



The error is 179,983,507.878, which is almost half the error of the straight-line model. This is good; however, it comes with a price. We now have a more complex function, meaning that we have one more parameter to tune inside `polyfit()`. The fitted polynomial is as follows:

$$f(x) = 0.0105322215 * x^{**2} - 5.26545650 * x + 1974.76082$$

So, if more complexity gives better results, why not increase the complexity even more? Let's try it for degree 3, 10, and 100.



The more complex the data gets, the curves capture it and make it fit better. The errors seem to tell the same story.

```
Error d=1: 317,389,767.339778
Error d=2: 179,983,507.878179
Error d=3: 139,350,144.031725
Error d=10: 121,942,326.363461
Error d=100: 109,318,004.475556
```

However, taking a closer look at the fitted curves, we start to wonder whether they also capture the true process that generated this data. Framed differently, do our models correctly represent the underlying mass behavior of customers visiting our website? Looking at the polynomial of degree 10 and 100, we see wildly oscillating behavior. It seems that the models are fitted too much to the data. So much that it is now capturing not only the underlying process but also the noise. This is called **overfitting**.

At this point, we have the following choices:

- Selecting one of the fitted polynomial models.
- Switching to another more complex model class; splines?
- Thinking differently about the data and starting again.

Of the five fitted models, the first-order model clearly is too simple, and the models of order 10 and 100 are clearly overfitting. Only the second- and third-order models seem to somehow match the data. However, if we extrapolate them at both borders, we see them going berserk.

Switching to a more complex class also seems to be the wrong way to go about it. What arguments would back which class? At this point, we realize that we probably have not completely understood our data.

Stepping back to go forward – another look at our data

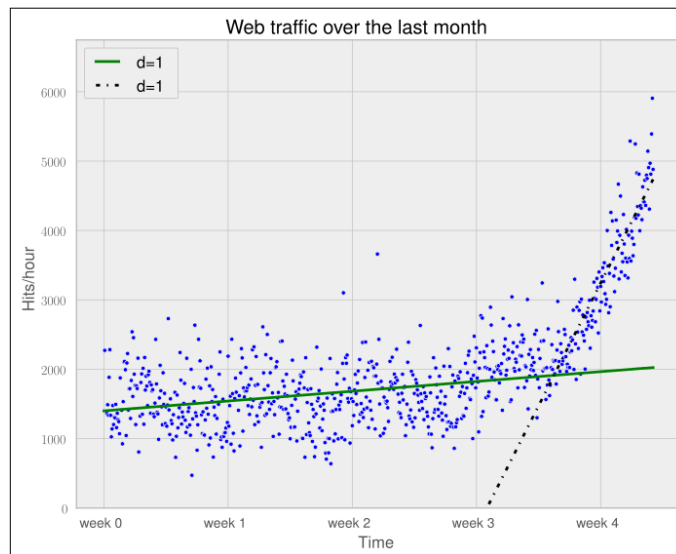
So, we step back and take another look at the data. It seems that there is an inflection point between weeks 3 and 4. So let us separate the data and train two lines using week 3.5 as a separation point. We train the first line with the data up to week 3, and the second line with the remaining data.

```
inflection = 3.5*7*24 # calculate the inflection point in hours
xa = x[:inflection] # data before the inflection point
ya = y[:inflection]
xb = x[inflection:] # data after
yb = y[inflection:]

fa = sp.poly1d(sp.polyfit(xa, ya, 1))
fb = sp.poly1d(sp.polyfit(xb, yb, 1))

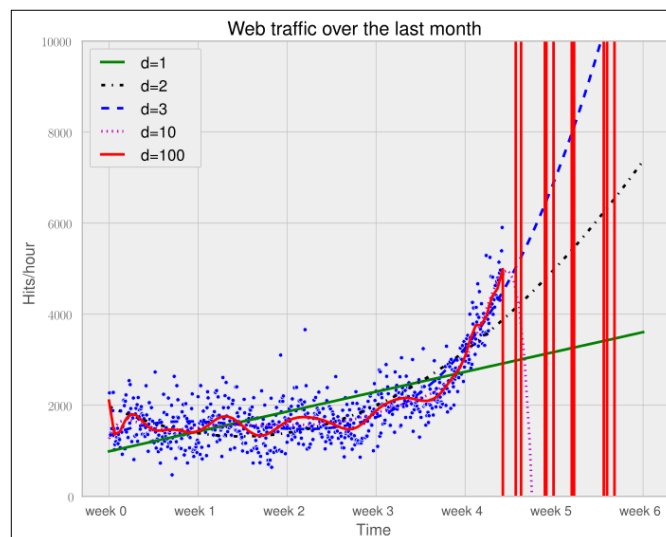
fa_error = error(fa, xa, ya)
fb_error = error(fb, xb, yb)
print("Error inflection=%f" % (fa + fb_error))
Error inflection=156,639,407.701523
```

Plotting the two models for the two data ranges gives the following chart:



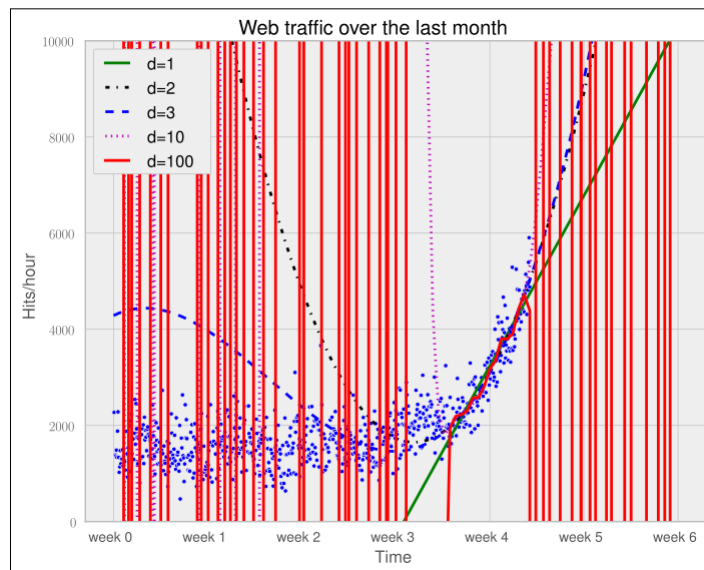
Clearly, the combination of these two lines seems to be a much better fit to the data than anything we have modeled before. But still, the combined error is higher than the higher-order polynomials. Can we trust the error at the end?

Asked differently, why do we trust the straight line fitted only at the last week of our data more than any of the more complex models? It is because we assume that it will capture future data better. If we plot the models into the future, we see how right we are ($d=1$ is again our initially straight line).



The models of degree 10 and 100 don't seem to expect a bright future for our startup. They tried so hard to model the given data correctly that they are clearly useless to extrapolate further. This is called overfitting. On the other hand, the lower-degree models do not seem to be capable of capturing the data properly. This is called underfitting.

So let us play fair to the models of degree 2 and above and try out how they behave if we fit them *only* to the data of the last week. After all, we believe that the last week says more about the future than the data before. The result can be seen in the following psychedelic chart, which shows even more clearly how bad the problem of overfitting is:



Still, judging from the errors of the models when trained only on the data from week 3.5 and after, we should still choose the most complex one.

```
Error d=1: 22143941.107618
Error d=2: 19768846.989176
Error d=3: 19766452.361027
Error d=10: 18949339.348539
Error d=100: 16915159.603877
```

Training and testing

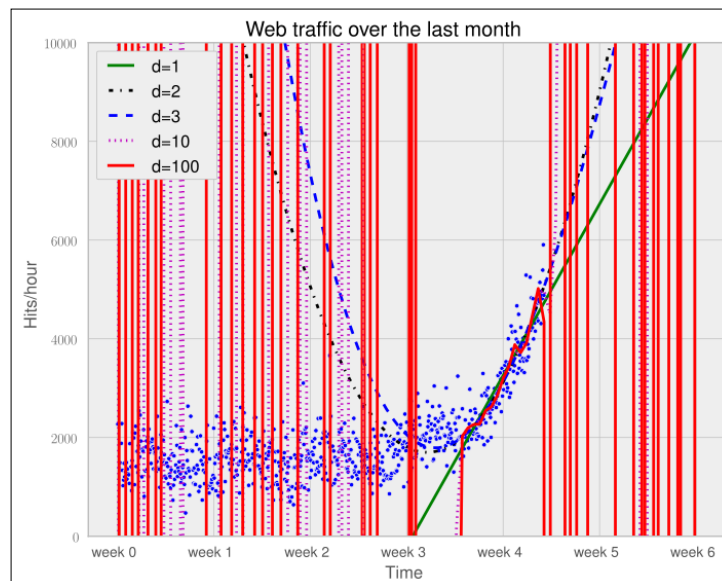
If only we had some data from the future that we could use to measure our models against, we should be able to judge our model choice only on the resulting approximation error.

Although we cannot look into the future, we can and should simulate a similar effect by holding out a part of our data. Let us remove, for instance, a certain percentage of the data and train on the remaining one. Then we use the hold-out data to calculate the error. As the model has been trained not knowing the hold-out data, we should get a more realistic picture of how the model will behave in the future.

The test errors for the models trained only on the time after the inflection point now show a completely different picture.

```
Error d=1: 7,917,335.831122
Error d=2: 6,993,880.348870
Error d=3: 7,137,471.177363
Error d=10: 8,805,551.189738
Error d=100: 10,877,646.621984
```

The result can be seen in the following chart:



It seems we finally have a clear winner. The model with degree 2 has the lowest test error, which is the error when measured using data that the model did not see during training. And this is what lets us trust that we won't get bad surprises when future data arrives.

Answering our initial question

Finally, we have arrived at a model that we think represents the underlying process best; it is now a simple task of finding out when our infrastructure will reach 100,000 requests per hour. We have to calculate when our model function reaches the value 100,000.

Having a polynomial of degree 2, we could simply compute the inverse of the function and calculate its value at 100,000. Of course, we would like to have an approach that is applicable to any model function easily.

This can be done by subtracting 100,000 from the polynomial, which results in another polynomial, and finding the root of it. SciPy's `optimize` module has the `fsolve` function to achieve this when providing an initial starting position. Let `fbt2` be the winning polynomial of degree 2:

```
>>> print(fbt2)
      2
0.08844 x - 97.31 x + 2.853e+04
>>> print(fbt2-100000)
      2
0.08844 x - 97.31 x - 7.147e+04

>>> from scipy.optimize import fsolve
>>> reached_max = fsolve(fbt2-100000, 800)/(7*24)
>>> print("100,000 hits/hour expected at week %f" % reached_max[0])
100,000 hits/hour expected at week 9.827613
```

Our model tells us that given the current user behavior and traction of our startup, it will take another month until we have reached our threshold capacity.

Of course, there is a certain uncertainty involved with our prediction. To get the real picture, you can draw in more sophisticated statistics to find out about the variance that we have to expect when looking farther and further into the future.

And then there are the user and underlying user behavior dynamics that we cannot model accurately. However, at this point we are fine with the current predictions. After all, we can prepare all the time-consuming actions now. If we then monitor our web traffic closely, we will see in time when we have to allocate new resources.

Summary

Congratulations! You just learned two important things. Of these, the most important one is that as a typical machine learning operator, you will spend most of your time understanding and refining the data – exactly what we just did in our first tiny machine learning example. And we hope that the example helped you to start switching your mental focus from algorithms to data. Later, you learned how important it is to have the correct experiment setup, and that it is vital to not mix up training and testing.

Admittedly, the use of polynomial fitting is not the coolest thing in the machine learning world. We have chosen it so as not to distract you with the coolness of some shiny algorithm, which encompasses the two most important points we just summarized above.

So, let's move to the next chapter, in which we will dive deep into SciKits-learn, the marvelous machine learning toolkit, give an overview of different types of learning, and show you the beauty of feature engineering.

2

Learning How to Classify with Real-world Examples

Can a machine distinguish between flower species based on images? From a machine learning perspective, we approach this problem by having the machine learn how to perform this task based on examples of each species so that it can classify images where the species are not marked. This process is called **classification** (or **supervised learning**), and is a classic problem that goes back a few decades.

We will explore small datasets using a few simple algorithms that we can implement manually. The goal is to be able to understand the basic principles of classification. This will be a solid foundation to understanding later chapters as we introduce more complex methods that will, by necessity, rely on code written by others.

The Iris dataset

The Iris dataset is a classic dataset from the 1930s; it is one of the first modern examples of statistical classification.

The setting is that of Iris flowers, of which there are multiple species that can be identified by their morphology. Today, the species would be defined by their genomic signatures, but in the 1930s, DNA had not even been identified as the carrier of genetic information.

The following four attributes of each plant were measured:

- Sepal length
- Sepal width
- Petal length
- Petal width

In general, we will call any measurement from our data as **features**.

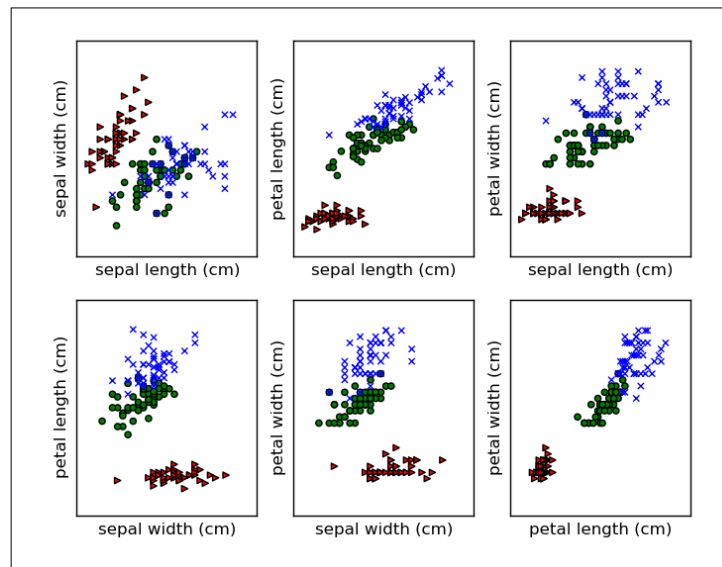
Additionally, for each plant, the species was recorded. The question now is: if we saw a new flower out in the field, could we make a good prediction about its species from its measurements?

This is the **supervised learning** or **classification** problem; given labeled examples, we can design a rule that will eventually be applied to other examples. This is the same setting that is used for spam classification; given the examples of spam and ham (non-spam e-mail) that the user gave the system, can we determine whether a new, incoming message is spam or not?

For the moment, the Iris dataset serves our purposes well. It is small (150 examples, 4 features each) and can easily be visualized and manipulated.

The first step is visualization

Because this dataset is so small, we can easily plot all of the points and all two-dimensional projections on a page. We will thus build intuitions that can then be extended to datasets with many more dimensions and datapoints. Each subplot in the following screenshot shows all the points projected into two of the dimensions. The outlying group (triangles) are the Iris Setosa plants, while Iris Versicolor plants are in the center (circle) and Iris Virginica are indicated with "x" marks. We can see that there are two large groups: one is of Iris Setosa and another is a mixture of Iris Versicolor and Iris Virginica.



We are using Matplotlib; it is the most well-known plotting package for Python. We present the code to generate the top-left plot. The code for the other plots is similar to the following code:

```
from matplotlib import pyplot as plt
from sklearn.datasets import load_iris
import numpy as np

# We load the data with load_iris from sklearn
data = load_iris()
features = data['data']
feature_names = data['feature_names']
target = data['target']

for t,marker,c in zip(xrange(3),">ox","rgb"):
    # We plot each class on its own to get different colored markers
    plt.scatter(features[target == t,0],
                features[target == t,1],
                marker=marker,
                c=c)
```

Building our first classification model

If the goal is to separate the three types of flower, we can immediately make a few suggestions. For example, the petal length seems to be able to separate Iris Setosa from the other two flower species on its own. We can write a little bit of code to discover where the cutoff is as follows:

```
length = features[:, 2]
# use numpy operations to get setosa features
is_setosa = (labels == 'setosa')
# This is the important step:
max_setosa = length[is_setosa].max()
min_non_setosa = length[~is_setosa].min()
print('Maximum of setosa: {0}'.format(max_setosa))
print('Minimum of others: {0}'.format(min_non_setosa))
```

This prints **1.9** and **3.0**. Therefore, we can build a simple model: *if the petal length is smaller than two, this is an Iris Setosa flower; otherwise, it is either Iris Virginica or Iris Versicolor.*

```
if features[:,2] < 2: print 'Iris Setosa'
else: print 'Iris Virginica or Iris Versicolour'
```

This is our first model, and it works very well in that it separates the Iris Setosa flowers from the other two species without making any mistakes.

What we had here was a simple structure; a simple threshold on one of the dimensions. Then we searched for the best dimension threshold. We performed this visually and with some calculation; machine learning happens when we write code to perform this for us.

The example where we distinguished Iris Setosa from the other two species was very easy. However, we cannot immediately see what the best threshold is for distinguishing Iris Virginica from Iris Versicolor. We can even see that we will never achieve perfect separation. We can, however, try to do it the best possible way. For this, we will perform a little computation.

We first select only the non-Setosa features and labels:

```
features = features[~is_setosa]
labels = labels[~is_setosa]
virginica = (labels == 'virginica')
```

Here we are heavily using NumPy operations on the arrays. `is_setosa` is a Boolean array, and we use it to select a subset of the other two arrays, `features` and `labels`. Finally, we build a new Boolean array, `virginica`, using an equality comparison on labels.

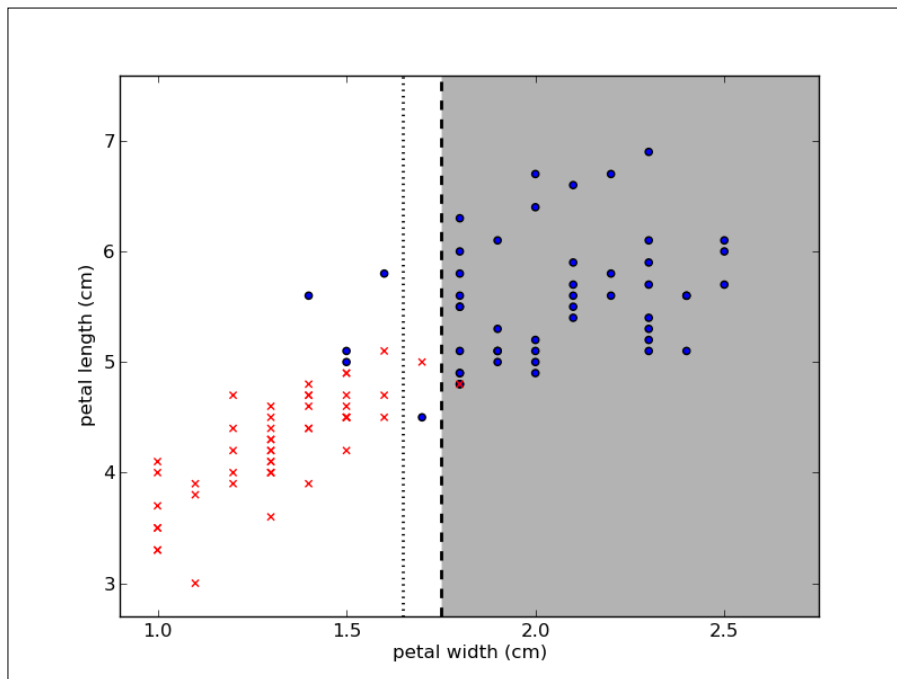
Now, we run a loop over all possible features and thresholds to see which one results in better accuracy. *Accuracy* is simply the fraction of examples that the model classifies correctly:

```
best_acc = -1.0
for fi in xrange(features.shape[1]):
    # We are going to generate all possible threshold for this feature
    thresh = features[:,fi].copy()
    thresh.sort()
    # Now test all thresholds:
    for t in thresh:
        pred = (features[:,fi] > t)
        acc = (pred == virginica).mean()
        if acc > best_acc:
            best_acc = acc
            best_fi = fi
            best_t = t
```

The last few lines select the best model. First we compare the predictions, `pred`, with the actual labels, `virginica`. The little trick of computing the mean of the comparisons gives us the fraction of correct results, the accuracy. At the end of the `for` loop, all possible thresholds for all possible features have been tested, and the `best_fi` and `best_t` variables hold our model. To apply it to a new example, we perform the following:

```
if example[best_fi] > t: print 'virginica'
else: print 'versicolor'
```

What does this model look like? If we run it on the whole data, the best model that we get is split on the petal length. We can visualize the decision boundary. In the following screenshot, we see two regions: one is white and the other is shaded in grey. Anything that falls in the white region will be called Iris Virginica and anything that falls on the shaded side will be classified as Iris Versicolor:



In a threshold model, the decision boundary will always be a line that is parallel to one of the axes. The plot in the preceding screenshot shows the decision boundary and the two regions where the points are classified as either white or grey. It also shows (as a dashed line) an alternative threshold that will achieve exactly the same accuracy. Our method chose the first threshold, but that was an arbitrary choice.

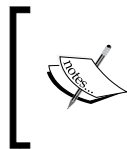
Evaluation – holding out data and cross-validation

The model discussed in the preceding section is a simple model; it achieves 94 percent accuracy on its training data. However, this evaluation may be overly optimistic. We used the data to define what the threshold would be, and then we used the same data to evaluate the model. Of course, the model will perform better than anything else we have tried on this dataset. The logic is circular.

What we really want to do is estimate the ability of the model to generalize to new instances. We should measure its performance in instances that the algorithm has not seen at training. Therefore, we are going to do a more rigorous evaluation and use held-out data. For this, we are going to break up the data into two blocks: on one block, we'll train the model, and on the other – the one we held out of training – we'll test it. The output is as follows:

```
Training error was 96.0%.
Testing error was 90.0% (N = 50).
```

The result of the testing data is lower than that of the training error. This may surprise an inexperienced machine learner, but it is expected and typical. To see why, look back at the plot that showed the decision boundary. See if some of the examples close to the boundary were not there or if one of the ones in between the two lines was missing. It is easy to imagine that the boundary would then move a little bit to the right or to the left so as to put them on the "wrong" side of the border.



The error on the training data is called a **training error** and is always an overly optimistic estimate of how well your algorithm is doing. We should always measure and report the **testing error**; the error on a collection of examples that were not used for training.

These concepts will become more and more important as the models become more complex. In this example, the difference between the two errors is not very large. When using a complex model, it is possible to get 100 percent accuracy in training and do no better than random guessing on testing!

One possible problem with what we did previously, which was to hold off data from training, is that we only used part of the data (in this case, we used half of it) for training. On the other hand, if we use too little data for testing, the error estimation is performed on a very small number of examples. Ideally, we would like to use all of the data for training and all of the data for testing as well.

We can achieve something quite similar by **cross-validation**. One extreme (but sometimes useful) form of cross-validation is leave-one-out. We will take an example out of the training data, learn a model without this example, and then see if the model classifies this example correctly:

```

error = 0.0
for ei in range(len(features)):
    # select all but the one at position 'ei':
    training = np.ones(len(features), bool)
    training[ei] = False
    testing = ~training
    model = learn_model(features[training], virginica[training])
    predictions = apply_model(features[testing],
                              virginica[testing], model)
    error += np.sum(predictions != virginica[testing])

```

At the end of this loop, we will have tested a series of models on all the examples. However, there is no circularity problem because each example was tested on a model that was built without taking the model into account. Therefore, the overall estimate is a reliable estimate of how well the models would generalize.

The major problem with leave-one-out cross-validation is that we are now being forced to perform 100 times more work. In fact, we must learn a whole new model for each and every example, and this will grow as our dataset grows.

We can get most of the benefits of leave-one-out at a fraction of the cost by using x -fold cross-validation; here, " x " stands for a small number, say, five. In order to perform five-fold cross-validation, we break up the data in five groups, that is, five folds.

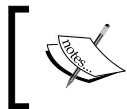
Then we learn five models, leaving one fold out of each. The resulting code will be similar to the code given earlier in this section, but here we leave 20 percent of the data out instead of just one element. We test each of these models on the left out fold and average the results:

Dataset	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
1	Test	Train	Train	Train	Train
2	Train	Test	Train	Train	Train
3	Train	Train	Test	Train	Train
4	Train	Train	Train	Test	Train
5	Train	Train	Train	Train	Test

The preceding figure illustrates this process for five blocks; the dataset is split into five pieces. Then for each fold, you hold out one of the blocks for testing and train on the other four. You can use any number of folds you wish. Five or ten fold is typical; it corresponds to training with 80 or 90 percent of your data and should already be close to what you would get from using all the data. In an extreme case, if you have as many folds as datapoints, you can simply perform leave-one-out cross-validation.

When generating the folds, you need to be careful to keep them balanced. For example, if all of the examples in one fold come from the same class, the results will not be representative. We will not go into the details of how to do this because the machine learning packages will handle it for you.

We have now generated several models instead of just one. So, what *final model* do we return and use for the new data? The simplest solution is now to use a single overall model on all your training data. The cross-validation loop gives you an estimate of how well this model should generalize.



A cross-validation schedule allows you to use all your data to estimate if your methods are doing well. At the end of the cross-validation loop, you can use all your data to train a final model.

Although it was not properly recognized when machine learning was starting out, nowadays it is seen as a very bad sign to even discuss the training error of a classification system. This is because the results can be very misleading. We always want to measure and compare either the error on a held-out dataset or the error estimated using a cross-validation schedule.

Building more complex classifiers

In the previous section, we used a very simple model: a threshold on one of the dimensions. Throughout this book, you will see many other types of models, and we're not even going to cover everything that is out there.

What makes up a classification model? We can break it up into three parts:

- **The structure of the model:** In this, we use a threshold on a single feature.
- **The search procedure:** In this, we try every possible combination of feature and threshold.
- **The loss function:** Using the loss function, we decide which of the possibilities is less bad (because we can rarely talk about the perfect solution). We can use the training error or just define this point the other way around and say that we want the best accuracy. Traditionally, people want the loss function to be minimum.

We can play around with these parts to get different results. For example, we can attempt to build a threshold that achieves minimal training error, but we will only test three values for each feature: the mean value of the features, the mean plus one standard deviation, and the mean minus one standard deviation. This could make sense if testing each value was very costly in terms of computer time (or if we had millions and millions of datapoints). Then the exhaustive search we used would be infeasible, and we would have to perform an approximation like this.

Alternatively, we might have different loss functions. It might be that one type of error is much more costly than another. In a medical setting, false negatives and false positives are not equivalent. A **false negative** (when the result of a test comes back negative, but that is false) might lead to the patient not receiving treatment for a serious disease. A **false positive** (when the test comes back positive even though the patient does not actually have that disease) might lead to additional tests for confirmation purposes or unnecessary treatment (which can still have costs, including side effects from the treatment). Therefore, depending on the exact setting, different trade-offs can make sense. At one extreme, if the disease is fatal and treatment is cheap with very few negative side effects, you want to minimize the false negatives as much as you can. With spam filtering, we may face the same problem; incorrectly deleting a non-spam e-mail can be very dangerous for the user, while letting a spam e-mail through is just a minor annoyance.



What the **cost function** should be is always dependent on the exact problem you are working on. When we present a general-purpose algorithm, we often focus on minimizing the number of mistakes (achieving the highest accuracy). However, if some mistakes are more costly than others, it might be better to accept a lower overall accuracy to minimize overall costs.

Finally, we can also have other classification structures. A simple threshold rule is very limiting and will only work in the very simplest cases, such as with the Iris dataset.

A more complex dataset and a more complex classifier

We will now look at a slightly more complex dataset. This will motivate the introduction of a new classification algorithm and a few other ideas.

Learning about the Seeds dataset

We will now look at another agricultural dataset; it is still small, but now too big to comfortably plot exhaustively as we did with Iris. This is a dataset of the measurements of wheat seeds. Seven features are present, as follows:

- Area (A)
- Perimeter (P)
- Compactness ($C = 4\pi A/P^2$)
- Length of kernel
- Width of kernel
- Asymmetry coefficient
- Length of kernel groove

There are three classes that correspond to three wheat varieties: Canadian, Koma, and Rosa. As before, the goal is to be able to classify the species based on these morphological measurements.

Unlike the Iris dataset, which was collected in the 1930s, this is a very recent dataset, and its features were automatically computed from digital images.

This is how image pattern recognition can be implemented: you can take images in digital form, compute a few relevant features from them, and use a generic classification system. In a later chapter, we will work through the computer vision side of this problem and compute features in images. For the moment, we will work with the features that are given to us.



UCI Machine Learning Dataset Repository

The **University of California at Irvine (UCI)** maintains an online repository of machine learning datasets (at the time of writing, they are listing 233 datasets). Both the Iris and Seeds dataset used in this chapter were taken from there.

The repository is available online:
<http://archive.ics.uci.edu/ml/>

Features and feature engineering

One interesting aspect of these features is that the compactness feature is not actually a new measurement, but a function of the previous two features, area and perimeter. It is often very useful to derive new combined features. This is a general area normally termed **feature engineering**; it is sometimes seen as less glamorous than algorithms, but it may matter more for performance (a simple algorithm on well-chosen features will perform better than a fancy algorithm on not-so-good features).

In this case, the original researchers computed the "compactness", which is a typical feature for shapes (also called "roundness"). This feature will have the same value for two kernels, one of which is twice as big as the other one, but with the same shape. However, it will have different values for kernels that are very round (when the feature is close to one) as compared to kernels that are elongated (when the feature is close to zero).

The goals of a good feature are to simultaneously vary with what matters and be invariant with what does not. For example, compactness does not vary with size but varies with the shape. In practice, it might be hard to achieve both objectives perfectly, but we want to approximate this ideal.

You will need to use background knowledge to intuit which will be good features. Fortunately, for many problem domains, there is already a vast literature of possible features and feature types that you can build upon. For images, all of the previously mentioned features are typical, and computer vision libraries will compute them for you. In text-based problems too, there are standard solutions that you can mix and match (we will also see this in a later chapter). Often though, you can use your knowledge of the specific problem to design a specific feature.

Even before you have data, you must decide which data is worthwhile to collect. Then, you need to hand all your features to the machine to evaluate and compute the best classifier.

A natural question is whether or not we can select good features automatically. This problem is known as **feature selection**. There are many methods that have been proposed for this problem, but in practice, very simple ideas work best. It does not make sense to use feature selection in these small problems, but if you had thousands of features, throwing out most of them might make the rest of the process much faster.

Nearest neighbor classification

With this dataset, even if we just try to separate two classes using the previous method, we do not get very good results. Let me introduce , therefore, a new classifier: the nearest neighbor classifier.

If we consider that each example is represented by its features (in mathematical terms, as a point in N-dimensional space), we can compute the distance between examples. We can choose different ways of computing the distance, for example:

```
def distance(p0, p1):
    'Computes squared euclidean distance'
    return np.sum( (p0-p1)**2)
```

Now when classifying, we adopt a simple rule: given a new example, we look at the dataset for the point that is closest to it (its nearest neighbor) and look at its label:

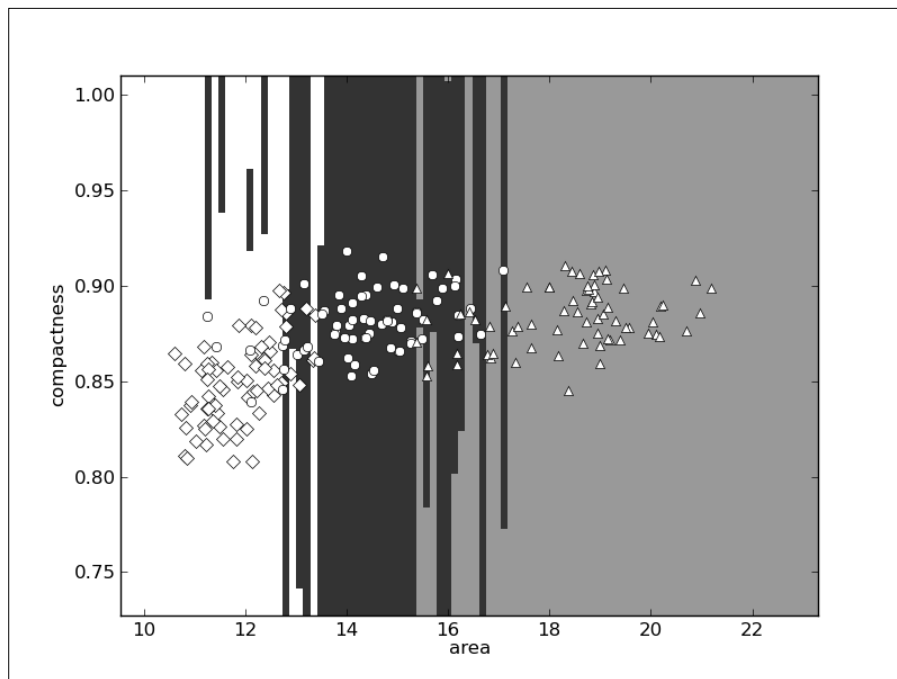
```
def nn_classify(training_set, training_labels, new_example):
    dists = np.array([distance(t, new_example)
                      for t in training_set])
    nearest = dists.argmin()
    return training_labels[nearest]
```

In this case, our model involves saving all of the training data and labels and computing everything at classification time. A better implementation would be to actually index these at learning time to speed up classification, but this implementation is a complex algorithm.

Now, note that this model performs perfectly on its training data! For each point, its closest neighbor is itself, and so its label matches perfectly (unless two examples have exactly the same features but different labels, which can happen). Therefore, it is essential to test using a cross-validation protocol.

Using ten folds for cross-validation for this dataset with this algorithm, we obtain 88 percent accuracy. As we discussed in the earlier section, the cross-validation accuracy is lower than the training accuracy, but this is a more credible estimate of the performance of the model.

We will now examine the decision boundary. For this, we will be forced to simplify and look at only two dimensions (just so that we can plot it on paper).



In the preceding screenshot, the Canadian examples are shown as diamonds, Kama seeds as circles, and Rosa seeds as triangles. Their respective areas are shown as white, black, and grey. You might be wondering why the regions are so horizontal, almost weirdly so. The problem is that the x axis (area) ranges from 10 to 22 while the y axis (compactness) ranges from 0.75 to 1.0. This means that a small change in x is actually much larger than a small change in y . So, when we compute the distance according to the preceding function, we are, for the most part, only taking the x axis into account.

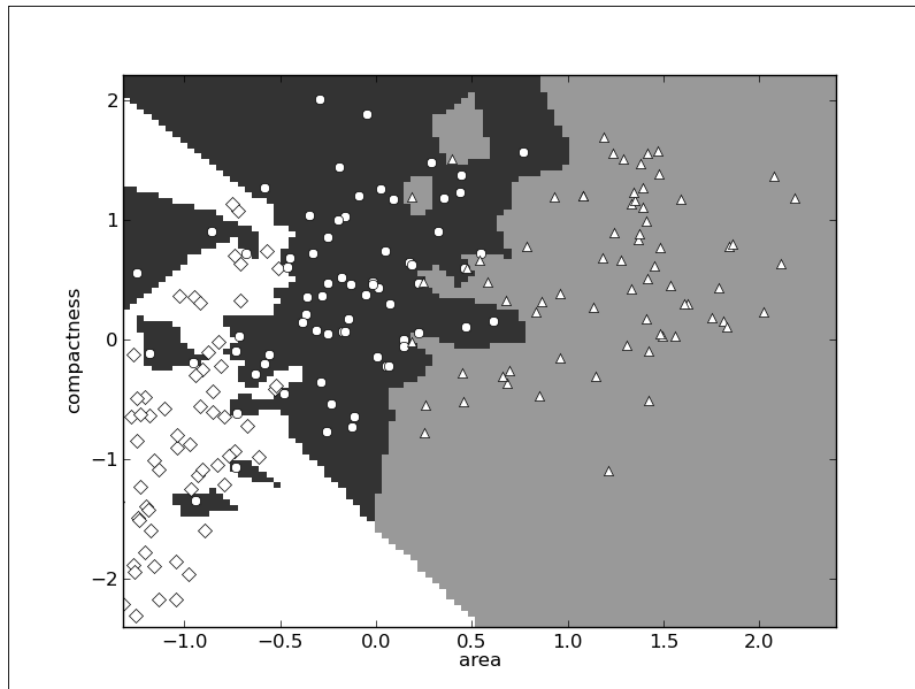
If you have a physics background, you might have already noticed that we had been summing up lengths, areas, and dimensionless quantities, mixing up our units (which is something you never want to do in a physical system). We need to normalize all of the features to a common scale. There are many solutions to this problem; a simple one is to normalize to Z-scores. The Z-score of a value is how far away from the mean it is in terms of units of standard deviation. It comes down to this simple pair of operations:

```
# subtract the mean for each feature:
features -= features.mean(axis=0)
# divide each feature by its standard deviation
features /= features.std(axis=0)
```

Independent of what the original values were, after Z-scoring, a value of zero is the mean and positive values are above the mean and negative values are below it.

Now every feature is in the same unit (technically, every feature is now dimensionless; it has no units) and we can mix dimensions more confidently. In fact, if we now run our nearest neighbor classifier, we obtain 94 percent accuracy!

Look at the decision space again in two dimensions; it looks as shown in the following screenshot:



The boundaries are now much more complex and there is interaction between the two dimensions. In the full dataset, everything is happening in a seven-dimensional space that is very hard to visualize, but the same principle applies: where before a few dimensions were dominant, now they are all given the same importance.

The nearest neighbor classifier is simple, but sometimes good enough. We can generalize it to a *k-nearest* neighbor classifier by considering not just the closest point but the *k* closest points. All *k* neighbors vote to select the label. *k* is typically a small number, such as 5, but can be larger, particularly if the dataset is very large.

Binary and multiclass classification

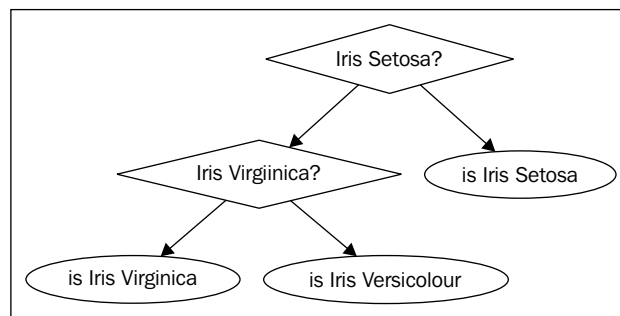
The first classifier we saw, the threshold classifier, was a simple binary classifier (the result is either one class or the other as a point is either above the threshold or it is not). The second classifier we used, the nearest neighbor classifier, was a naturally multiclass classifier (the output can be one of several classes).

It is often simpler to define a simple binary method than one that works on multiclass problems. However, we can reduce the multiclass problem to a series of binary decisions. This is what we did earlier in the Iris dataset in a haphazard way; we observed that it was easy to separate one of the initial classes and focused on the other two, reducing the problem to two binary decisions:

- Is it an Iris Setosa (yes or no)?
- If no, check whether it is an Iris Virginica (yes or no).

Of course, we want to leave this sort of reasoning to the computer. As usual, there are several solutions to this multiclass reduction.

The simplest is to use a series of "one classifier versus the rest of the classifiers". For each possible label ℓ , we build a classifier of the type "is this ℓ or something else?". When applying the rule, exactly one of the classifiers would say "yes" and we would have our solution. Unfortunately, this does not always happen, so we have to decide how to deal with either multiple positive answers or no positive answers.



Alternatively, we can build a classification tree. Split the possible labels in two and build a classifier that asks "should this example go to the left or the right bin?" We can perform this splitting recursively until we obtain a single label. The preceding diagram depicts the tree of reasoning for the Iris dataset. Each diamond is a single binary classifier. It is easy to imagine we could make this tree larger and encompass more decisions. This means that any classifier that can be used for binary classification can also be adapted to handle any number of classes in a simple way.

There are many other possible ways of turning a binary method into a multiclass one. There is no single method that is clearly better in all cases. However, which one you use normally does not make much of a difference to the final result.

Most classifiers are binary systems while many real-life problems are naturally multiclass. Several simple protocols reduce a multiclass problem to a series of binary decisions and allow us to apply the binary models to our multiclass problem.

Summary

In a sense, this was a very theoretical chapter, as we introduced generic concepts with simple examples. We went over a few operations with a classic dataset. This, by now, is considered a very small problem. However, it has the advantage that we were able to plot it out and see what we were doing in detail. This is something that will be lost when we move on to problems with many dimensions and many thousands of examples. The intuitions we gained here will all still be valid.

Classification means generalizing from examples to build a model (that is, a rule that can automatically be applied to new, unclassified objects). It is one of the fundamental tools in machine learning, and we will see many more examples of this in forthcoming chapters.

We also learned that the training error is a misleading, over-optimistic estimate of how well the model does. We must, instead, evaluate it on testing data that was not used for training. In order to not waste too many examples in testing, a cross-validation schedule can get us the best of both worlds (at the cost of more computation).

We also had a look at the problem of feature engineering. Features are not something that is predefined for you, but choosing and designing features is an integral part of designing a machine-learning pipeline. In fact, it is often the area where you can get the most improvements in accuracy as better data beats fancier methods. The chapters on computer vision and text-based classification will provide examples for these specific settings.

In this chapter, we wrote all of our own code (except when we used NumPy, of course). This will not be the case for the next few chapters, but we needed to build up intuitions on simple cases to illustrate the basic concepts.

The next chapter looks at how to proceed when your data does not have predefined classes for classification.

3

Clustering – Finding Related Posts

In the previous chapter, we have learned how to find classes or categories of individual data points. With a handful of training data items that were paired with their respective classes, we learned a model that we can now use to classify future data items. We called this supervised learning, as the learning was guided by a teacher; in our case the teacher had the form of correct classifications.

Let us now imagine that we do not possess those labels by which we could learn the classification model. This could be, for example, because they were too expensive to collect. What could we have done in that case?

Well, of course, we would not be able to learn a classification model. Still, we could find some pattern within the data itself. This is what we will do in this chapter, where we consider the challenge of a "question and answer" website. When a user browses our site looking for some particular information, the search engine will most likely point him/her to a specific answer. To improve the user experience, we now want to show all related questions with their answers. If the presented answer is not what he/she was looking for, he/she can easily see the other available answers and hopefully stay on our site.

The naive approach would be to take the post, calculate its similarity to all other posts, and display the top N most similar posts as links on the page. This will quickly become very costly. Instead, we need a method that quickly finds all related posts.

We will achieve this goal in this chapter using clustering. This is a method of arranging items so that similar items are in one cluster and dissimilar items are in distinct ones. The tricky thing that we have to tackle first is how to turn text into something on which we can calculate similarity. With such a measurement for similarity, we will then proceed to investigate how we can leverage that to quickly arrive at a cluster that contains similar posts. Once there, we will only have to check out those documents that also belong to that cluster. To achieve this, we will introduce the marvelous Scikit library, which comes with diverse machine-learning methods that we will also use in the following chapters.

Measuring the relatedness of posts

From the machine learning point of view, raw text is useless. Only if we manage to transform it into meaningful numbers, can we feed it into our machine-learning algorithms such as clustering. The same is true for more mundane operations on text, such as similarity measurement.

How not to do it

One text similarity measure is the Levenshtein distance, which also goes by the name edit distance. Let's say we have two words, "machine" and "mchiene". The similarity between them can be expressed as the minimum set of edits that are necessary to turn one word into the other. In this case, the edit distance would be 2, as we have to add an "a" after "m" and delete the first "e". This algorithm is, however, quite costly, as it is bound by the product of the lengths of the first and second words.

Looking at our posts, we could cheat by treating the whole word as characters and performing the edit distance calculation on the word level. Let's say we have two posts (let's concentrate on the title for the sake of simplicity), "How to format my hard disk" and "Hard disk format problems"; we would have an edit distance of five (removing "how", "to", "format", "my", and then adding "format" and "problems" at the end). Therefore, one could express the difference between two posts as the number of words that have to be added or deleted so that one text morphs into the other. Although we could speed up the overall approach quite a bit, the time complexity stays the same.

Even if it would have been fast enough, there is another problem. The post above the word "format" accounts for an edit distance of two (deleting it first, then adding it). So our distance doesn't seem to be robust enough to take word reordering into account.

How to do it

More robust than edit distance is the so-called **bag-of-word** approach. It uses simple word counts as its basis. For each word in the post, its occurrence is counted and noted in a vector. Not surprisingly, this step is also called vectorization. The vector is typically huge as it contains as many elements as the words that occur in the whole dataset. Take for instance two example posts with the following word counts:

Word	Occurrences in Post 1	Occurrences in Post 2
disk	1	1
format	1	1
how	1	0
hard	1	1
my	1	0
problems	0	1
to	1	0

The columns Post 1 and Post 2 can now be treated as simple vectors. We could simply calculate the Euclidean distance between the vectors of all posts and take the nearest one (too slow, as we have just found out). As such, we can use them later in the form of feature vectors in the following clustering steps:

1. Extract the salient features from each post and store it as a vector per post.
2. Compute clustering on the vectors.
3. Determine the cluster for the post in question.
4. From this cluster, fetch a handful of posts that are different from the post in question. This will increase diversity.

However, there is some more work to be done before we get there, and before we can do that work, we need some data to work on.

Preprocessing – similarity measured as similar number of common words

As we have seen previously, the bag-of-word approach is both fast and robust. However, it is not without challenges. Let's dive directly into them.

Converting raw text into a bag-of-words

We do not have to write a custom code for counting words and representing those counts as a vector. Scikit's `CountVectorizer` does the job very efficiently. It also has a very convenient interface. Scikit's functions and classes are imported via the `sklearn` package as follows:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer(min_df=1)
```

The parameter `min_df` determines how `CountVectorizer` treats words that are not used frequently (minimum document frequency). If it is set to an integer, all words occurring less than that value will be dropped. If it is a fraction, all words that occur less than that fraction of the overall dataset will be dropped. The parameter `max_df` works in a similar manner. If we print the instance, we see what other parameters Scikit provides together with their default values:

```
>>> print(vectorizer)
CountVectorizer(analyzer=word, binary=False, charset=utf-8,
charset_error=strict, dtype=<type 'long'>, input=content,
lowercase=True, max_df=1.0, max_features=None, max_n=None,
min_df=1, min_n=None, ngram_range=(1, 1), preprocessor=None,
stop_words=None, strip_accents=None, token_pattern=(?u)\b\w\w+\b,
tokenizer=None, vocabulary=None)
```

We see that, as expected, the counting is done at word level (`analyzer=word`) and the words are determined by the regular expression pattern `token_pattern`. It would, for example, tokenize "cross-validated" into "cross" and "validated". Let us ignore the other parameters for now.

```
>>> content = ["How to format my hard disk", " Hard disk format
problems "]
>>> X = vectorizer.fit_transform(content)
>>> vectorizer.get_feature_names()
[u'disk', u'format', u'hard', u'how', u'my', u'problems', u'to']
```

The vectorizer has detected seven words for which we can fetch the counts individually:

```
>>> print(X.toarray().transpose())
array([[1, 1],
       [1, 1],
       [1, 1],
       [1, 0],
       [1, 0],
       [0, 1],
       [1, 0]], dtype=int64)
```

This means that the first sentence contains all the words except for "problems", while the second contains all except "how", "my", and "to". In fact, these are exactly the same columns as seen in the previous table. From X , we can extract a feature vector that we can use to compare the two documents with each other.

First we will start with a naive approach to point out some preprocessing peculiarities we have to account for. So let us pick a random post, for which we will then create the count vector. We will then compare its distance to all the count vectors and fetch the post with the smallest one.

Counting words

Let us play with the toy dataset consisting of the following posts:

Post filename	Post content
01.txt	This is a toy post about machine learning. Actually, it contains not much interesting stuff.
02.txt	Imaging databases can get huge.
03.txt	Most imaging databases save images permanently.
04.txt	Imaging databases store images.
05.txt	Imaging databases store images. Imaging databases store images. Imaging databases store images.

In this post dataset, we want to find the most similar post for the short post "imaging databases".

Assuming that the posts are located in the folder `DIR`, we can feed `CountVectorizer` with it as follows:

```
>>> posts = [open(os.path.join(DIR, f)).read() for f in
os.listdir(DIR)]
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer(min_df=1)
```

We have to notify the vectorizer about the full dataset so that it knows upfront what words are to be expected, as shown in the following code:

```
>>> X_train = vectorizer.fit_transform(posts)
>>> num_samples, num_features = X_train.shape
>>> print("#samples: %d, #features: %d" % (num_samples,
num_features)) #samples: 5, #features: 25
```



```

...     post = posts[i]

...     if post==new_post:
...         continue
...     post_vec = X_train.getrow(i)
...     d = dist(post_vec, new_post_vec)
...     print "=== Post %i with dist=%.2f: %s"%(i, d, post)
...     if d<best_dist:
...         best_dist = d
...         best_i = i
>>> print("Best post is %i with dist=%.2f"%(best_i, best_dist))

=== Post 0 with dist=4.00: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.
=== Post 1 with dist=1.73: Imaging databases provide storage
capabilities.
=== Post 2 with dist=2.00: Most imaging databases safe images
permanently.
=== Post 3 with dist=1.41: Imaging databases store data.
=== Post 4 with dist=5.10: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 3 with dist=1.41

```

Congratulations! We have our first similarity measurement. Post 0 is most dissimilar from our new post. Quite understandably, it does not have a single word in common with the new post. We can also understand that Post 1 is very similar to the new post, but not to the winner, as it contains one word more than Post 3 that is not contained in the new post.

Looking at posts 3 and 4, however, the picture is not so clear any more. Post 4 is the same as Post 3, duplicated three times. So, it should also be of the same similarity to the new post as Post 3.

Printing the corresponding feature vectors explains the reason:

```

>>> print(X_train.getrow(3).toarray())
[[0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]]
>>> print(X_train.getrow(4).toarray())
[[0 0 0 0 3 3 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0]]

```

Obviously, using only the counts of the raw words is too simple. We will have to normalize them to get vectors of unit length.

Normalizing the word count vectors

We will have to extend `dist_raw` to calculate the vector distance, not on the raw vectors but on the normalized ones instead:

```
>>> def dist_norm(v1, v2):
...     v1_normalized = v1/sp.linalg.norm(v1.toarray())
...     v2_normalized = v2/sp.linalg.norm(v2.toarray())
...     delta = v1_normalized - v2_normalized
...     return sp.linalg.norm(delta.toarray())
```

This leads to the following similarity measurement:

```
=== Post 0 with dist=1.41: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.
=== Post 1 with dist=0.86: Imaging databases provide storage
capabilities.
=== Post 2 with dist=0.92: Most imaging databases save images
permanently.
=== Post 3 with dist=0.77: Imaging databases store data.
=== Post 4 with dist=0.77: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 3 with dist=0.77
```

This looks a bit better now. Post 3 and Post 4 are calculated as being equally similar. One could argue whether that much repetition would be a delight to the reader, but from the point of counting the words in the posts, this seems to be right.

Removing less important words

Let us have another look at Post 2. Of its words that are not in the new post, we have "most", "safe", "images", and "permanently". They are actually quite different in the overall importance to the post. Words such as "most" appear very often in all sorts of different contexts, and words such as this are called stop words. They do not carry as much information, and thus should not be weighed as much as words such as "images", that don't occur often in different contexts. The best option would be to remove all words that are so frequent that they do not help to distinguish between different texts. These words are called stop words.

As this is such a common step in text processing, there is a simple parameter in `CountVectorizer` to achieve this, as follows:

```
>>> vectorizer = CountVectorizer(min_df=1, stop_words='english')
```

If you have a clear picture of what kind of stop words you would want to remove, you can also pass a list of them. Setting `stop_words` to "english" will use a set of 318 English stop words. To find out which ones they are, you can use `get_stop_words()`:

```
>>> sorted(vectorizer.get_stop_words())[0:20]
['a', 'about', 'above', 'across', 'after', 'afterwards', 'again',
 'against', 'all', 'almost', 'alone', 'along', 'already', 'also',
 'although', 'always', 'am', 'among', 'amongst', 'amoungst']
```

The new word list is seven words lighter:

```
[u'actually', u'capabilities', u'contains', u'data', u'databases',
 u'images', u'imaging', u'interesting', u'learning', u'machine',
 u'permanently', u'post', u'provide', u'safe', u'storage', u'store',
 u'stuff', u'toy']
```

Without stop words, we arrive at the following similarity measurement:

```
=== Post 0 with dist=1.41: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.
=== Post 1 with dist=0.86: Imaging databases provide storage
capabilities.
=== Post 2 with dist=0.86: Most imaging databases safe images
permanently.
=== Post 3 with dist=0.77: Imaging databases store data.
=== Post 4 with dist=0.77: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 3 with dist=0.77
```

Post 2 is now on par with Post 1. Overall, it has, however, not changed much as our posts are kept short for demonstration purposes. It will become vital when we look at real-world data.

Stemming

One thing is still missing. We count similar words in different variants as different words. Post 2, for instance, contains "imaging" and "images". It would make sense to count them together. After all, it is the same concept they are referring to.

We need a function that reduces words to their specific word stem. Scikit does not contain a stemmer by default. With the **Natural Language Toolkit (NLTK)**, we can download a free software toolkit, which provides a stemmer that we can easily plug into `CountVectorizer`.

Installing and using NLTK

How to install NLTK on your operating system is described in detail at <http://nltk.org/install.html>. Basically, you will need to install the two packages NLTK and PyYAML.

To check whether your installation was successful, open a Python interpreter and type the following:

```
>>> import nltk
```



You will find a very nice tutorial for NLTK in the book *Python Text Processing with NLTK 2.0 Cookbook*. To play a little bit with a stemmer, you can visit the accompanied web page <http://text-processing.com/demo/stem/>.

NLTK comes with different stemmers. This is necessary, because every language has a different set of rules for stemming. For English, we can take `SnowballStemmer`.

```
>>> import nltk.stem
>>> s = nltk.stem.SnowballStemmer('english')
>>> s.stem("graphics")
u'graphic'
>>> s.stem("imaging")
u'imag'
>>> s.stem("image")
u'imag'
>>> s.stem("imagination")u'imagin'
>>> s.stem("imagine")
u'imagin'
```



Note that stemming does not necessarily have to result into valid English words.

It also works with verbs as follows:

```
>>> s.stem("buys")
u'buy'
>>> s.stem("buying")
u'buy'
>>> s.stem("bought")
u'bought'
```

Extending the vectorizer with NLTK's stemmer

We need to stem the posts before we feed them into `CountVectorizer`. The class provides several hooks with which we could customize the preprocessing and tokenization stages. The preprocessor and tokenizer can be set in the constructor as parameters. We do not want to place the stemmer into any of them, because we would then have to do the tokenization and normalization by ourselves. Instead, we overwrite the method `build_analyzer` as follows:

```
>>> import nltk.stem
>>> english_stemmer = nltk.stem.SnowballStemmer('english')
>>> class StemmedCountVectorizer(CountVectorizer):
...     def build_analyzer(self):
...         analyzer = super(StemmedCountVectorizer, self).build_
analyzer()
...         return lambda doc: (english_stemmer.stem(w) for w in
analyzer(doc))
>>> vectorizer = StemmedCountVectorizer(min_df=1, stop_
words='english')
```

This will perform the following steps for each post:

1. Lower casing the raw post in the preprocessing step (done in the parent class).
2. Extracting all individual words in the tokenization step (done in the parent class).
3. Converting each word into its stemmed version.

As a result, we now have one feature less, because "images" and "imaging" collapsed to one. The set of feature names looks like the following:

```
[u'actual', u'capabl', u'contain', u'data', u'databas', u'imag',
u'interest', u'learn', u'machin', u'perman', u'post', u'provid',
u'safe', u'storag', u'store', u'stuff', u'toy']
```

Running our new stemmed vectorizer over our posts, we see that collapsing "imaging" and "images" reveals that Post 2 is actually the most similar post to our new post, as it contains the concept "imag" twice:

```
=== Post 0 with dist=1.41: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.
=== Post 1 with dist=0.86: Imaging databases provide storage
capabilities.
=== Post 2 with dist=0.63: Most imaging databases safe images
permanently.
```

```
=== Post 3 with dist=0.77: Imaging databases store data.
=== Post 4 with dist=0.77: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 2 with dist=0.63
```

Stop words on steroids

Now that we have a reasonable way to extract a compact vector from a noisy textual post, let us step back for a while to think about what the feature values actually mean.

The feature values simply count occurrences of terms in a post. We silently assumed that higher values for a term also mean that the term is of greater importance to the given post. But what about, for instance, the word "subject", which naturally occurs in each and every single post? Alright, we could tell `CountVectorizer` to remove it as well by means of its `max_df` parameter. We could, for instance, set it to 0.9 so that all words that occur in more than 90 percent of all posts would be always ignored. But what about words that appear in 89 percent of all posts? How low would we be willing to set `max_df`? The problem is that however we set it, there will always be the problem that some terms are just more discriminative than others.

This can only be solved by counting term frequencies for every post, and in addition, discounting those that appear in many posts. In other words, we want a high value for a given term in a given value if that term occurs often in that particular post and very rarely anywhere else.

This is exactly what **term frequency – inverse document frequency (TF-IDF)** does; TF stands for the counting part, while IDF factors in the discounting. A naive implementation would look like the following:

```
>>> import scipy as sp
>>> def tfidf(term, doc, docset):
...     tf = float(doc.count(term))/sum(doc.count(w) for w in docset)
...     idf = math.log(float(len(docset))/(len([doc for doc in docset
...         if term in doc])))
...     return tf * idf
```

For the following document set, `docset`, consisting of three documents that are already tokenized, we can see how the terms are treated differently, although all appear equally often per document:

```
>>> a, abb, abc = ["a"], ["a", "b", "b"], ["a", "b", "c"]
>>> D = [a, abb, abc]
>>> print(tfidf("a", a, D))
```

```
0.0
>>> print(tfidf("b", abb, D))
0.270310072072
>>> print(tfidf("a", abc, D))
0.0
>>> print(tfidf("b", abc, D))
0.135155036036
>>> print(tfidf("c", abc, D))
0.366204096223
```

We see that `a` carries no meaning for any document since it is contained everywhere. `b` is more important for the document `abb` than for `abc` as it occurs there twice.

In reality, there are more corner cases to handle than the above example does. Thanks to Scikit, we don't have to think of them, as they are already nicely packaged in `TfidfVectorizer`, which is inherited from `CountVectorizer`. Sure enough, we don't want to miss our stemmer:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> class StemmedTfidfVectorizer(TfidfVectorizer):
...     def build_analyzer(self):
...         analyzer = super(TfidfVectorizer,
...                           self).build_analyzer()
...         return lambda doc: (
...             english_stemmer.stem(w) for w in analyzer(doc))
>>> vectorizer = StemmedTfidfVectorizer(min_df=1,
...                                     stop_words='english', charset_error='ignore')
```

The resulting document vectors will not contain counts any more. Instead, they will contain the individual TF-IDF values per term.

Our achievements and goals

Our current text preprocessing phase includes the following steps:

1. Tokenizing the text.
2. Throwing away words that occur way too often to be of any help in detecting relevant posts.

3. Throwing away words that occur so seldom that there is only a small chance that they occur in future posts.
4. Counting the remaining words.
5. Calculating TF-IDF values from the counts, considering the whole text corpus.

Again we can congratulate ourselves. With this process, we are able to convert a bunch of noisy text into a concise representation of feature values.

But, as simple and as powerful as the bag-of-words approach with its extensions is, it has some drawbacks that we should be aware of. They are as follows:

- It does not cover word relations. With the previous vectorization approach, the text "Car hits wall" and "Wall hits car" will both have the same feature vector.
- It does not capture negations correctly. For instance, the text "I will eat ice cream" and "I will not eat ice cream" will look very similar by means of their feature vectors, although they contain quite the opposite meaning. This problem, however, can be easily changed by not only counting individual words, also called unigrams, but also considering bigrams (pairs of words) or trigrams (three words in a row).
- It totally fails with misspelled words. Although it is clear to the readers that "database" and "databas" convey the same meaning, our approach will treat them as totally different words.

For brevity's sake, let us nevertheless stick with the current approach, which we can now use to efficiently build clusters from.

Clustering

Finally, we have our vectors that we believe capture the posts to a sufficient degree. Not surprisingly, there are many ways to group them together. Most clustering algorithms fall into one of the two methods, flat and hierarchical clustering.

Flat clustering divides the posts into a set of clusters without relating the clusters to each other. The goal is simply to come up with a partitioning such that all posts in one cluster are most similar to each other while being dissimilar from the posts in all other clusters. Many flat clustering algorithms require the number of clusters to be specified up front.

In hierarchical clustering, the number of clusters does not have to be specified. Instead, the hierarchical clustering creates a hierarchy of clusters. While similar posts are grouped into one cluster, similar clusters are again grouped into one uber-cluster. This is done recursively, until only one cluster is left, which contains everything. In this hierarchy, one can then choose the desired number of clusters. However, this comes at the cost of lower efficiency.

Scikit provides a wide range of clustering approaches in the package `sklearn.cluster`. You can get a quick overview of the advantages and drawbacks of each of them at <http://scikit-learn.org/dev/modules/clustering.html>.

In the following section, we will use the flat clustering method, KMeans, and play a bit with the desired number of clusters.

KMeans

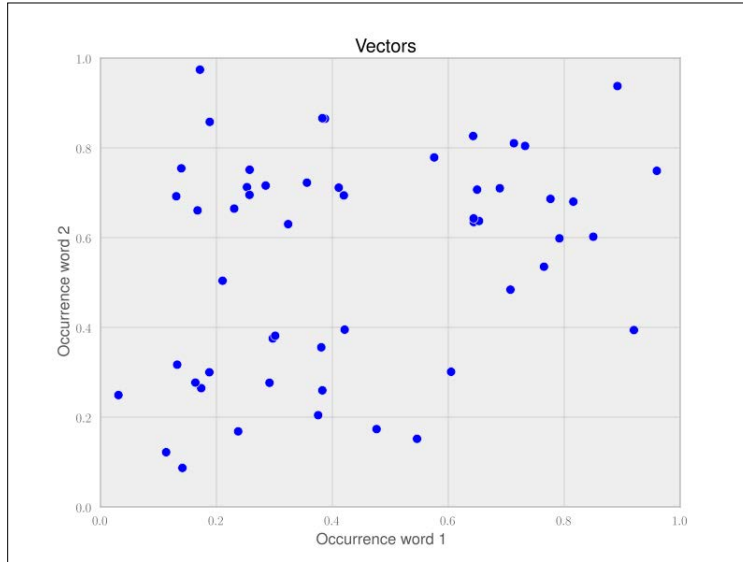
KMeans is the most widely used flat clustering algorithm. After it is initialized with the desired number of clusters, `num_clusters`, it maintains that number of so-called cluster centroids. Initially, it would pick any of the `num_clusters` posts and set the centroids to their feature vector. Then it would go through all other posts and assign them the nearest centroid as their current cluster. Then it will move each centroid into the middle of all the vectors of that particular class. This changes, of course, the cluster assignment. Some posts are now nearer to another cluster. So it will update the assignments for those changed posts. This is done as long as the centroids move a considerable amount. After some iterations, the movements will fall below a threshold and we consider clustering to be converged.



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

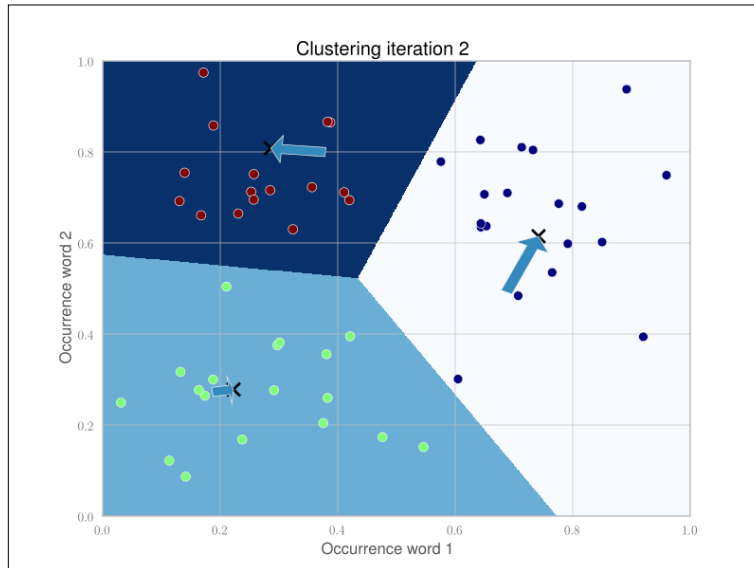
Let us play this through with a toy example of posts containing only two words. Each point in the following chart represents one document:



After running one iteration of KMeans, that is, taking any two vectors as starting points, assigning labels to the rest, and updating the cluster centers to be the new center point of all points in that cluster, we get the following clustering:



Because the cluster centers are moved, we have to reassign the cluster labels and recalculate the cluster centers. After iteration 2, we get the following clustering:



The arrows show the movements of the cluster centers. After five iterations in this example, the cluster centers don't move noticeably any more (Scikit's tolerance threshold is 0.0001 by default).

After the clustering has settled, we just need to note down the cluster centers and their identity. When each new document comes in, we have to vectorize and compare it with all the cluster centers. The cluster center with the smallest distance to our new post vector belongs to the cluster we will assign to the new post.

Getting test data to evaluate our ideas on

In order to test clustering, let us move away from the toy text examples and find a dataset that resembles the data we are expecting in the future so that we can test our approach. For our purpose, we need documents on technical topics that are already grouped together so that we can check whether our algorithm works as expected when we apply it later to the posts we hope to receive.

One standard dataset in machine learning is the 20newsgroup dataset, which contains 18,826 posts from 20 different newsgroups. Among the groups' topics are technical ones such as `comp.sys.mac.hardware` or `sci.crypt` as well as more politics- and religion-related ones such as `talk.politics.guns` or `soc.religion.christian`. We will restrict ourselves to the technical groups. If we assume each newsgroup is one cluster, we can nicely test whether our approach of finding related posts works.

The dataset can be downloaded from <http://people.csail.mit.edu/jrennie/20Newsgroups>. Much more simple, however, is to download it from MLComp at <http://mlcomp.org/datasets/379> (free registration required). Scikit already contains custom loaders for that dataset and rewards you with very convenient data loading options.

The dataset comes in the form of a ZIP file, `dataset-379-20news-18828_WJQIG.zip`, which we have to unzip to get the folder 379, which contains the datasets. We also have to notify Scikit about the path containing that data directory. It contains a metadata file and three directories `test`, `train`, and `raw`. The `test` and `train` directories split the whole dataset into 60 percent of training and 40 percent of testing posts. For convenience, the `dataset` module also contains the function `fetch_20newsgroups`, which downloads that data into the desired directory.



The website <http://mlcomp.org> is used for comparing machine-learning programs on diverse datasets. It serves two purposes: finding the right dataset to tune your machine-learning program and exploring how other people use a particular dataset. For instance, you can see how well other people's algorithms performed on particular datasets and compare them.

Either you set the environment variable `MLCOMP_DATASETS_HOME` or you specify the path directly with the `mlcomp_root` parameter when loading the dataset as follows:

```
>>> import sklearn.datasets
>>> MLCOMP_DIR = r"D:\data"
>>> data = sklearn.datasets.load_mlcomp("20news-18828", mlcomp_
root=MLCOMP_DIR)
>>> print(data filenames)
array(['D:\\data\\379\\raw\\comp.graphics\\1190-38614',
      'D:\\data\\379\\raw\\comp.graphics\\1383-38616',
      'D:\\data\\379\\raw\\alt.atheism\\487-53344',
      ...,
      'D:\\data\\379\\raw\\rec.sport.hockey\\10215-54303',
      'D:\\data\\379\\raw\\sci.crypt\\10799-15660',
```

```

        'D:\\data\\379\\raw\\comp.os.ms-windows.misc\\2732-10871'],
        dtype='|S68')
>>> print(len(data filenames))
18828
>>> data.target_names
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.
ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x', 'misc.
forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.
sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space',
'soc.religion.christian', 'talk.politics.guns', 'talk.politics.
mideast', 'talk.politics.misc', 'talk.religion.misc']

```

We can choose among training and test sets as follows:

```

>>> train_data = sklearn.datasets.load_mlcomp("20news-18828", "train",
mlcomp_root=MLCOMP_DIR)
>>> print(len(train_data filenames))
13180
>>> test_data = sklearn.datasets.load_mlcomp("20news-18828",
"test", mlcomp_root=MLCOMP_DIR)
>>> print(len(test_data filenames))
5648

```

For simplicity's sake, we will restrict ourselves to only some newsgroups so that the overall experimentation cycle is shorter. We can achieve this with the `categories` parameter as follows:

```

>>> groups = ['comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.
ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x', 'sci.
space']
>>> train_data = sklearn.datasets.load_mlcomp("20news-18828", "train",
mlcomp_root=MLCOMP_DIR, categories=groups)
>>> print(len(train_data filenames))
3414

```

Clustering posts

You must have already noticed one thing – real data is noisy. The newsgroup dataset is no exception. It even contains invalid characters that will result in `UnicodeDecodeError`.

We have to tell the vectorizer to ignore them:

```

>>> vectorizer = StemmedTfidfVectorizer(min_df=10, max_df=0.5,
...                                     stop_words='english', charset_error='ignore')
>>> vectorized = vectorizer.fit_transform(dataset.data)

```

```
>>> num_samples, num_features = vectorized.shape
>>> print("#samples: %d, #features: %d" % (num_samples, num_features))
#samples: 3414, #features: 4331
```

We now have a pool of 3,414 posts and extracted for each of them a feature vector of 4,331 dimensions. That is what KMeans takes as input. We will fix the cluster size to 50 for this chapter and hope you are curious enough to try out different values as an exercise, as shown in the following code:

```
>>> num_clusters = 50
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=num_clusters, init='random', n_init=1,
               verbose=1)
>>> km.fit(vectorized)
```

That's it. After fitting, we can get the clustering information out of the members of `km`. For every vectorized post that has been fit, there is a corresponding integer label in `km.labels_`:

```
>>> km.labels_
array([33, 22, 17, ..., 14, 11, 39])
>>> km.labels_.shape
(3414,)
```

The cluster centers can be accessed via `km.cluster_centers_`.

In the next section we will see how we can assign a cluster to a newly arriving post using `km.predict`.

Solving our initial challenge

We now put everything together and demonstrate our system for the following new post that we assign to the variable `new_post`:

```
Disk drive problems. Hi, I have a problem with my hard disk.
After 1 year it is working only sporadically now.
I tried to format it, but now it doesn't boot any more.
Any ideas? Thanks.
```

As we have learned previously, we will first have to vectorize this post before we predict its label as follows:

```
>>> new_post_vec = vectorizer.transform([new_post])
>>> new_post_label = km.predict(new_post_vec)[0]
```

Now that we have the clustering, we do not need to compare `new_post_vec` to all post vectors. Instead, we can focus only on the posts of the same cluster. Let us fetch their indices in the original dataset:

```
>>> similar_indices = (km.labels_==new_post_label).nonzero()[0]
```

The comparison in the bracket results in a Boolean array, and `nonzero` converts that array into a smaller array containing the indices of the `True` elements.

Using `similar_indices`, we then simply have to build a list of posts together with their similarity scores as follows:

```
>>> similar = []
>>> for i in similar_indices:
...     dist = sp.linalg.norm((new_post_vec - vectorized[i]).toarray())
...     similar.append((dist, dataset.data[i]))
>>> similar = sorted(similar)
>>> print(len(similar))
44
```

We found 44 posts in the cluster of our post. To give the user a quick idea of what kind of similar posts are available, we can now present the most similar post (`show_at_1`), the least similar one (`show_at_3`), and an in-between post (`show_at_2`), all of which are from the same cluster as follows:

```
>>> show_at_1 = similar[0]
>>> show_at_2 = similar[len(similar)/2]
>>> show_at_3 = similar[-1]
```

The following table shows the posts together with their similarity values:

Position	Similarity	Excerpt from post
1	1.018	<p>BOOT PROBLEM with IDE controller</p> <p>Hi,</p> <p>I've got a Multi I/O card (IDE controller + serial/parallel interface) and two floppy drives (5 1/4, 3 1/2) and a Quantum ProDrive 80AT connected to it. I was able to format the hard disk, but I could not boot from it. I can boot from drive A: (which disk drive does not matter) but if I remove the disk from drive A and press the reset switch, the LED of drive A: continues to glow, and the hard disk is not accessed at all. I guess this must be a problem of either the Multi I/o card\nor floppy disk drive settings (jumper configuration?) Does someone have any hint what could be the reason for it. [...]</p>
2	1.294	<p>IDE Cable</p> <p>I just bought a new IDE hard drive for my system to go with the one I already had. My problem is this. My system only had a IDE cable for one drive, so I had to buy cable with two drive connectors on it, and consequently have to switch cables. The problem is, the new hard drive\'s manual refers to matching pin 1 on the cable with both pin 1 on the drive itself and pin 1 on the IDE card. But for the life of me I cannot figure out how to tell which way to plug in the cable to align these. Secondly, the cable has like a connector at two ends and one between them. I figure one end goes in the controller and then the other two go into the drives. Does it matter which I plug into the "master" drive and which into the "Slave"? any help appreciated [...]</p>

Position	Similarity	Excerpt from post
3	1.375	Conner CP3204F info please How to change the cluster size Wondering if somebody could tell me if we can change the cluster size of my IDE drive. Normally I can do it with Norton's Calibrat on MFM/RLL drives but dunno if I can on IDE too. [...]

It is interesting how the posts reflect the similarity measurement score. The first post contains all the salient words from our new post. The second one also revolves around hard disks, but lacks concepts such as formatting. Finally, the third one is only slightly related. Still, for all the posts, we would say that they belong to the same domain as that of the new post.

Another look at noise

We should not expect a perfect clustering, in the sense that posts from the same newsgroup (for example, `comp.graphics`) are also clustered together. An example will give us a quick impression of the noise that we have to expect:

```
>>> post_group = zip(dataset.data, dataset.target)
>>> z = (len(post[0]), post[0], dataset.target_names[post[1]]) for
post in post_group
>>> print(sorted(z)[5:7])
[(107, 'From: "kwansik kim" <kkim@cs.indiana.edu>\nSubject: Where
is FAQ ?\n\nWhere can I find it ?\n\nThanks, Kwansik\n\n', 'comp.
graphics'), (110, 'From: lioness@maple.circa.ufl.edu\nSubject: What is
3dO?\n\n\nSomeone please fill me in on what 3do.\n\nThanks,\n\nBH\n',
'comp.graphics')]
```

For both of these posts, there is no real indication that they belong to `comp.graphics`, considering only the wording that is left after the preprocessing step:

```
>>> analyzer = vectorizer.build_analyzer()
>>> list(analyzer(z[5][1]))
[u'kwansik', u'kim', u'kkim', u'cs', u'indiana', u'edu', u'subject',
u'faq', u'thank', u'kwansik']
>>> list(analyzer(z[6][1]))
[u'lioness', u'mapl', u'circa', u'ufl', u'edu', u'subject', u'3do',
u'3do', u'thank', u'bh']
```

This is only after tokenization, lower casing, and stop word removal. If we also subtract those words that will be later filtered out via `min_df` and `max_df`, which will be done later in `fit_transform`, it gets even worse:

```
>>> list(set(analyzer(z[5][1])).intersection(
        vectorizer.get_feature_names()))
[u'cs', u'faq', u'thank']
>>> list(set(analyzer(z[6][1])).intersection(
        vectorizer.get_feature_names()))
[u'bh', u'thank']
```

Furthermore, most of the words occur frequently in other posts as well, as we can check with the IDF scores. Remember that the higher the TF-IDF, the more discriminative a term is for a given post. And as IDF is a multiplicative factor here, a low value of it signals that it is not of great value in general:

```
>>> for term in ['cs', 'faq', 'thank', 'bh', 'thank']:
...     print('IDF(%s)=%.2f'%(term,
        vectorizer._tfidf.idf_[vectorizer.vocabulary_[term]]))
IDF(cs)=3.23
IDF(faq)=4.17
IDF(thank)=2.23
IDF(bh)=6.57
IDF(thank)=2.23
```

So, except for `bh`, which is close to the maximum overall IDF value of 6.74, the terms don't have much discriminative power. Understandably, posts from different newsgroups will be clustered together.

For our goal, however, this is no big deal, as we are only interested in cutting down the number of posts that we have to compare a new post to. After all, the particular newsgroup from where our training data came from is of no special interest.

Tweaking the parameters

So what about all the other parameters? Can we tweak them all to get better results?

Sure. We could, of course, tweak the number of clusters or play with the vectorizer's `max_features` parameter (you should try that!). Also, we could play with different cluster center initializations. There are also more exciting alternatives to KMeans itself. There are, for example, clustering approaches that also let you use different similarity measurements such as Cosine similarity, Pearson, or Jaccard. An exciting field for you to play.

But before you go there, you will have to define what you actually mean by "better". Scikit has a complete package dedicated only to this definition. The package is called `sklearn.metrics` and also contains a full range of different metrics to measure clustering quality. Maybe that should be the first place to go now, right into the sources of the metrics package.

Summary

That was a tough ride, from preprocessing over clustering to a solution that can convert noisy text into a meaningful concise vector representation that we can cluster. If we look at the efforts we had to do to finally be able to cluster, it was more than half of the overall task, but on the way, we learned quite a bit on text processing and how simple counting can get you very far in the noisy real-world data.

The ride has been made much smoother though, because of Scikit and its powerful packages. And there is more to explore. In this chapter we were scratching the surface of its capabilities. In the next chapters we will see more of its powers.

4

Topic Modeling

In the previous chapter we clustered texts into groups. This is a very useful tool, but it is not always appropriate. Clustering results in each text belonging to exactly one cluster. This book is about machine learning and Python. Should it be grouped with other Python-related works or with machine-related works? In the paper book age, a bookstore would need to make this decision when deciding where to stock it. In the Internet store age, however, the answer is that this book is both about machine learning and Python, and the book can be listed in both sections. We will, however, not list it in the food section.

In this chapter, we will learn methods that do not cluster objects, but put them into a small number of groups called topics. We will also learn how to derive between topics that are central to the text and others only that are vaguely mentioned (this book mentions plotting every so often, but it is not a central topic such as machine learning is). The subfield of machine learning that deals with these problems is called **topic modeling**.

Latent Dirichlet allocation (LDA)

LDA and LDA: unfortunately, there are two methods in machine learning with the initials LDA: latent Dirichlet allocation, which is a topic modeling method; and linear discriminant analysis, which is a classification method. They are completely unrelated, except for the fact that the initials LDA can refer to either. However, this can be confusing. Scikit-learn has a submodule, `sklearn.lda`, which implements linear discriminant analysis. At the moment, scikit-learn does not implement latent Dirichlet allocation.

The simplest topic model (on which all others are based) is **latent Dirichlet allocation (LDA)**. The mathematical ideas behind LDA are fairly complex, and we will not go into the details here.

For those who are interested and adventurous enough, a Wikipedia search will provide all the equations behind these algorithms at the following link:

http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation

However, we can understand that this is at a high level and there is a sort of fable which underlies these models. In this fable, there are topics that are fixed. This lacks clarity. Which documents?

For example, let's say we have only three topics at present:

- Machine learning
- Python
- Baking

Each topic has a list of words associated with it. This book would be a mixture of the first two topics, perhaps 50 percent each. Therefore, when we are writing it, we pick half of our words from the machine learning topic and half from the Python topic. In this model, the order of words does not matter.

The preceding explanation is a simplification of the reality; each topic assigns a probability to each word so that it is possible to use the word "flour" when the topic is either machine learning or baking, but more probable if the topic is baking.

Of course, we do not know what the topics are. Otherwise, this would be a different and much simpler problem. Our task right now is to take a collection of text and reverse engineer this fable in order to discover what topics are out there and also where each document belongs.

Building a topic model

Unfortunately, scikit-learn does not support latent Dirichlet allocation. Therefore, we are going to use the `gensim` package in Python. `Gensim` is developed by *Radim Řehůřek*, who is a machine learning researcher and consultant in the Czech Republic. We must start by installing it. We can achieve this by running one of the following commands:

```
pip install gensim
easy_install gensim
```

We are going to use an **Associated Press (AP)** dataset of news reports. This is a standard dataset, which was used in some of the initial work on topic models:

```
>>> from gensim import corpora, models, similarities
>>> corpus = corpora.BleiCorpus('./data/ap/ap.dat',
'/data/ap/vocab.txt')
```

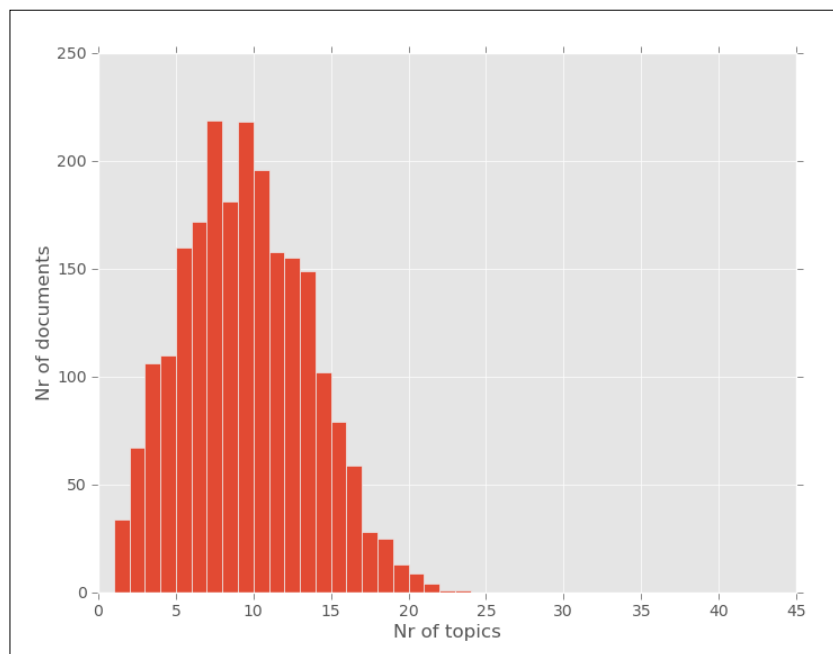
Corpus is just the preloaded list of words:

```
>>> model = models.ldamodel.LdaModel(  
    corpus,  
    num_topics=100,  
    id2word=corpus.id2word)
```

This one-step process will build a topic model. We can explore the topics in many ways. We can see the list of topics a document refers to by using the `model[doc]` syntax:

```
>>> topics = [model[c] for c in corpus]  
>>> print topics[0]  
[(3, 0.023607255776894751),  
 (13, 0.11679936618551275),  
 (19, 0.075935855202707139),  
 (92, 0.10781541687001292)]
```

I elided some of the output, but the format is a list of pairs (`topic_index`, `topic_weight`). We can see that only a few topics are used for each document. The topic model is a sparse model, as although there are many possible topics for each document, only a few of them are used. We can plot a histogram of the number of topics as shown in the following graph:





Sparsity means that while you may have large matrices and vectors, in principle, most of the values are zero (or so small that we can round them to zero as a good approximation). Therefore, only a few things are relevant at any given time.

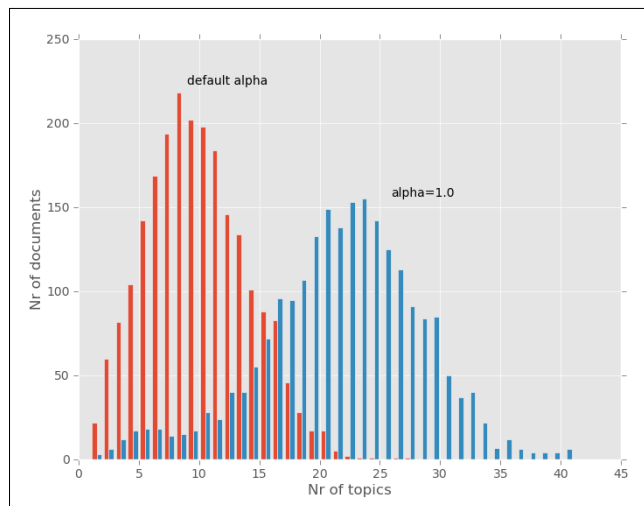
Often problems that seem too big to solve are actually feasible because the data is sparse. For example, even though one webpage can link to any other webpage, the graph of links is actually very sparse as each webpage will link to a very tiny fraction of all other webpages.

In the previous graph, we can see that about **150** documents have **5** topics, while the majority deal with around 10 to 12 of them. No document talks about more than 20 topics.

To a large extent, this is a function of the parameters used, namely the alpha parameter. The exact meaning of alpha is a bit abstract, but bigger values for alpha will result in more topics per document. Alpha needs to be positive, but is typically very small; usually smaller than one. By default, `gensim` will set alpha equal to $1.0 / \text{len}(\text{corpus})$, but you can set it yourself as follows:

```
>>> model = models.ldamodel.LdaModel(  
    corpus,  
    num_topics=100,  
    id2word=corpus.id2word,  
    alpha=1)
```

In this case, this is a larger alpha, which should lead to more topics per document. We could also use a smaller value. As we can see in the combined histogram given next, `gensim` behaves as we expected:



Now we can see that many documents touch upon 20 to 25 different topics.


What are these topics? Technically, they are multinomial distributions over words, which mean that they give each word in the vocabulary a probability. Words with high probability are more associated with that topic than words with lower probability.

Our brains aren't very good at reasoning with probability distributions, but we can readily make sense of a list of words. Therefore, it is typical to summarize topics with a the list of the most highly weighted words. Here are the first ten topics:

- dress military soviet president new state capt carlucci states leader stance government
- koch zambia lusaka one-party orange kochs party i government mayor new political
- human turkey rights abuses royal thompson threats new state wrote garden president
- bill employees experiments levin taxation federal measure legislation senate president whistleblowers sponsor
- ohio july drought jesus disaster percent hartford mississippi crops northern valley virginia
- united percent billion year president world years states people i bush news
- b hughes affidavit states united ounces squarefoot care delaying charged unrealistic bush
- yeutter dukakis bush convention farm subsidies uruguay percent secretary general i told
- Kashmir government people srinagar india dumps city two jammu-kashmir group moslem pakistan
- workers vietnamese irish wage immigrants percent bargaining last island police hutton I

However, topics are often just an intermediate tool to another end. Now that we have an estimate for each document about how much of that document comes from each topic, we can compare the documents in topic space. This simply means that instead of comparing word per word, we say that two documents are similar if they talk about the same topics.

This can be very powerful, as two text documents that share a few words may actually refer to the same topic. They may just refer to it using different constructions (for example, one may say the President of the United States while the other will use the name Barack Obama).

 Topic models are useful on their own to build visualizations and explore data. They are also very useful as an intermediate step in many other tasks.

At this point, we can redo the exercise we performed in the previous chapter and look for the most similar post, but by using the topics. Whereas previously we compared two documents by comparing their word vectors, we can now compare two documents by comparing their topic vectors.

For this, we are going to project the documents to the topic space. That is, we want to have a vector of topics that summarizes the document. Since the number of topics (100) is smaller than the number of possible words, we have reduced dimensionality. How to perform these types of dimensionality reduction in general is an important task in itself, and we have a chapter entirely devoted to this task. One additional computational advantage is that it is much faster to compare 100 vectors of topic weights than vectors of the size of the vocabulary (which will contain thousands of terms).

Using `gensim`, we saw before how to compute the topics corresponding to all documents in the corpus:

```
>>> topics = [model[c] for c in corpus]
>>> print topics[0]
[(3, 0.023607255776894751),
 (13, 0.11679936618551275),
 (19, 0.075935855202707139),
 (92, 0.10781541687001292)]
```

We will store all these topic counts in NumPy arrays and compute all pairwise distances:

```
>>> dense = np.zeros( (len(topics), 100), float)
>>> for ti,t in enumerate(topics):
```



```
...     for tj,v in t:
...         dense[ti,tj] = v
```

Now, `dense` is a matrix of topics. We can use the `pdist` function in SciPy to compute all pairwise distances. That is, with a single function call, we compute all the values of `sum((dense[ti] - dense[tj])**2)`:


```
>>> from scipy.spatial import distance
>>> pairwise = distance.squareform(distance.pdist(dense))
```

Now we employ one last little trick; we set the diagonal elements of the distance matrix to a high value (it just needs to be larger than the other values in the matrix):

```
>>> largest = pairwise.max()
>>> for ti in range(len(topics)):
>>>     pairwise[ti,ti] = largest+1
```

And we are done! For each document, we can look up the closest element easily:

```
>>> def closest_to(doc_id):
>>>     return pairwise[doc_id].argmin()
```

 The previous code would not work if we had not set the diagonal elements to a large value; the function would always return the same element as it is almost similar to itself (except in the weird case where two elements have exactly the same topic distribution, which is very rare unless they are exactly the same).

For example, here is the second document in the collection (the first document is very uninteresting, as the system returns a post stating that it is the most similar):

```
From: geb@cs.pitt.edu (Gordon Banks)
Subject: Re: request for information on "essential tremor" and Indrol?
In article <1q1tbnINNnfn@life.ai.mit.edu> sundar@ai.mit.edu writes:
Essential tremor is a progressive hereditary tremor that gets worse
when the patient tries to use the effected member. All limbs, vocal
cords, and head can be involved.  Inderal is a beta-blocker and is
usually effective in diminishing the tremor. Alcohol and mysoline are
also effective, but alcohol is too toxic to use as a treatment.
-----Gordon
Banks N3JXP      | "Skepticism is the chastity of the intellect, and
geb@cadre.dsl.pitt.edu | it is shameful to surrender it too soon."
-----
```

If we ask for the most similar document, `closest_to(1)`, we receive the following document:

```
From: geb@cs.pitt.edu (Gordon Banks)
```

Subject: Re: High Prolactin

In article <93088.112203JER4@psuvm.psu.edu> JER4@psuvm.psu.edu (John E. Rodway) writes:

>Any comments on the use of the drug Parlodel for high prolactin in the blood?

>It can suppress secretion of prolactin. Is useful in cases of galactorrhea. Some adenomas of the pituitary secret too much.

```
-----
Gordon Banks N3JXP      | "Skepticism is the chastity of the
intellect, and geb@cadre.dsl.pitt.edu | it is shameful to surrender
it too soon."
-----
```

We received a post by the same author discussing medications.

Modeling the whole of Wikipedia

While the initial LDA implementations could be slow, modern systems can work with very large collections of data. Following the documentation of `gensim`, we are going to build a topic model for the whole of the English language Wikipedia. This takes hours, but can be done even with a machine that is not too powerful. With a cluster of machines, we could make it go much faster, but we will look at that sort of processing in a later chapter.

First we download the whole Wikipedia dump from <http://dumps.wikimedia.org>. This is a large file (currently just over 9 GB), so it may take a while, unless your Internet connection is very fast. Then, we will index it with a `gensim` tool:

```
python -m gensim.scripts.make_wiki enwiki-latest-pages-articles.xml.bz2
wiki_en_output
```

Run the previous command on the command line, not on the Python shell. After a few hours, the indexing will be finished. Finally, we can build the final topic model. This step looks exactly like what we did for the small AP dataset. We first import a few packages:

```
>>> import logging, gensim
>>> logging.basicConfig(
    format='%(asctime)s : %(levelname)s : %(message)s',
    level=logging.INFO)
```

Now, we load the data that has been preprocessed:

```
>>> id2word =
gensim.corpora.Dictionary.load_from_text('wiki_en_output_wordids.txt')
>>> mm = gensim.corpora.MmCorpus('wiki_en_output_tfidf.mm')
```

Finally, we build the LDA model as before:

```
>>> model = gensim.models.ldamodel.LdaModel(  
    corpus=mm,  
    id2word=id2word,  
    num_topics=100,  
    update_every=1,  
    chunksize=10000,  
    passes=1)
```

This will again take a couple of hours (you will see the progress on your console, which can give you an indication of how long you still have to wait). Once it is done, you can save it to a file so you don't have to redo it all the time:

```
>>> model.save('wiki_lda.pkl')
```

If you exit your session and come back later, you can load the model again with:

```
>>> model = gensim.models.ldamodel.LdaModel.load('wiki_lda.pkl')
```

Let us explore some topics:

```
>>> topics = []  
>>> for doc in mm:  
    topics.append(model[doc])
```

We can see that this is still a sparse model even if we have many more documents than before (over 4 million as we are writing this):

```
>>> import numpy as np  
>>> lens = np.array([len(t) for t in topics])  
>>> print np.mean(lens)  
6.55842326445  
>>> print np.mean(lens <= 10)  
0.932382190219
```

So, the average document mentions 6.5 topics and 93 percent of them mention 10 or fewer.



If you have not seen the idiom before, it may be odd to take the mean of a comparison, but it is a direct way to compute a fraction. `np.mean(lens <= 10)` is taking the mean of an array of Booleans. The Booleans get interpreted as 0s and 1s in a numeric context. Therefore, the result is a number between 0 and 1, which is the fraction of ones. In this case, it is the fraction of elements of `lens`, which are less than or equal to 10.

If you are going to explore the topics yourself or build a visualization tool, you should probably try a few values and see which gives you the most useful or most appealing results.

However, there are a few methods that will automatically determine the number of topics for you depending on the dataset. One popular model is called the hierarchical Dirichlet process. Again, the full mathematical model behind it is complex and beyond the scope of this book, but the fable we can tell is that instead of having the topics be fixed a priori and our task being to reverse engineer the data to get them back, the topics themselves were generated along with the data. Whenever the writer was going to start a new document, he had the option of using the topics that already existed or creating a completely new one.

This means that the more documents we have, the more topics we will end up with. This is one of those statements that is unintuitive at first, but makes perfect sense upon reflection. We are learning topics, and the more examples we have, the more we can break them up. If we only have a few examples of news articles, then sports will be a topic. However, as we have more, we start to break it up into the individual modalities such as Hockey, Soccer, and so on. As we have even more data, we can start to tell nuances apart articles about individual teams and even individual players. The same is true for people. In a group of many different backgrounds, with a few "computer people", you might put them together; in a slightly larger group, you would have separate gatherings for programmers and systems managers. In the real world, we even have different gatherings for Python and Ruby programmers.

One of the methods for automatically determining the number of topics is called the **hierarchical Dirichlet process (HDP)**, and it is available in `gensim`. Using it is trivial. Taking the previous code for LDA, we just need to replace the call to `gensim.models.ldamodel.LdaModel` with a call to the `HdpModel` constructor as follows:

```
>>> hdp = gensim.models.hdpmodel.HdpModel(mm, id2word)
```

That's it (except it takes a bit longer to compute – there are no free lunches). Now, we can use this model as much as we used the LDA model, except that we did not need to specify the number of topics.

Summary

In this chapter, we discussed a more advanced form of grouping documents, which is more flexible than simple clustering as we allow each document to be present in more than one group. We explored the basic LDA model using a new package, `gensim`, but were able to integrate it easily into the standard Python scientific ecosystem.

Topic modeling was first developed and is easier to understand in the case of text, but in *Chapter 10, Computer Vision – Pattern Recognition*, we will see how some of these techniques may be applied to images as well. Topic models are very important in most of modern computer vision research. In fact, unlike the previous chapters, this chapter was very close to the cutting edge of research in machine learning algorithms. The original LDA algorithm was published in a scientific journal in 2003, but the method that `gensim` uses to be able to handle Wikipedia was only developed in 2010, and the HDP algorithm is from 2011. The research continues and you can find many variations and models with wonderful names such as the Indian buffet process (not to be confused with the Chinese restaurant process, which is a different model), or Pachinko allocation (Pachinko being a type of Japanese game, a cross between a slot-machine and pinball). Currently, they are still in the realm of research. In a few years, though, they might make the jump into the real world.

We have now gone over some of the major machine learning models such as classification, clustering, and topic modeling. In the next chapter, we go back to classification, but this time we will be exploring advanced algorithms and approaches.