4 Handling large data on a single computer

This chapter covers

- Working with large data sets on a single computer
- Working with Python libraries suitable for larger data sets
- Understanding the importance of choosing correct algorithms and data structures
- Understanding how you can adapt algorithms to work inside databases

What if you had so much data that it seems to outgrow you, and your techniques no longer seem to suffice? What do you do, surrender or adapt?

Luckily you chose to adapt, because you're still reading. This chapter introduces you to techniques and tools to handle larger data sets that are still manageable by a single computer if you adopt the right techniques.

This chapter gives you the tools to perform the classifications and regressions when the data no longer fits into the RAM (random access memory) of your computer, whereas chapter 3 focused on in-memory data sets. Chapter 5 will go a step further and teach you how to deal with data sets that require multiple computers to be processed. When we refer to *large data* in this chapter we mean data that causes problems to work with in terms of memory or speed but can still be handled by a single computer.

We start this chapter with an overview of the problems you face when handling large data sets. Then we offer three types of solutions to overcome these problems: adapt your algorithms, choose the right data structures, and pick the right tools. Data scientists aren't the only ones who have to deal with large data volumes, so you can apply general best practices to tackle the large data problem. Finally, we apply this knowledge to two case studies. The first case shows you how to detect malicious URLs, and the second case demonstrates how to build a recommender engine inside a database.

4.1 The problems you face when handling large data

A large volume of data poses new challenges, such as overloaded memory and algorithms that never stop running. It forces you to adapt and expand your repertoire of techniques. But even when you can perform your analysis, you should take care of issues such as I/O (input/output) and CPU starvation, because these can cause speed issues. Figure 4.1 shows a mind map that will gradually unfold as we go through the steps: problems, solutions, and tips.

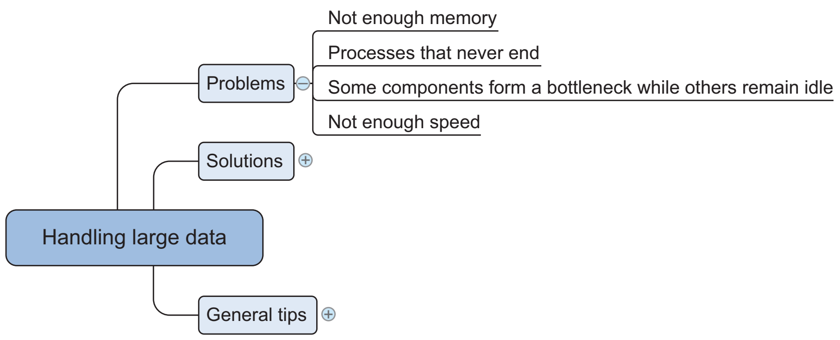


Figure 4.1 Overview of problems encountered when working with more data than can fit in memory

A computer only has a limited amount of RAM. When you try to squeeze more data into this memory than actually fits, the OS will start swapping out memory blocks to disks, which is far less efficient than having it all in memory. But only a few algorithms are designed to handle large data sets; most of them load the whole data set into memory at once, which causes the out-of-memory error. Other algorithms need to hold multiple copies of the data in memory or store intermediate results. All of these aggravate the problem.

Even when you cure the memory issues, you may need to deal with another limited resource: *time*. Although a computer may think you live for millions of years, in reality you won't (unless you go into cryostasis until your PC is done). Certain algorithms don't take time into account; they'll keep running forever. Other algorithms can't end in a reasonable amount of time when they need to process only a few megabytes of data.

A third thing you'll observe when dealing with large data sets is that components of your computer can start to form a bottleneck while leaving other systems idle. Although this isn't as severe as a never-ending algorithm or out-of-memory errors, it still incurs a serious cost. Think of the cost savings in terms of person days and computing infrastructure for CPU starvation. Certain programs don't feed data fast enough to the processor because they have to read data from the hard drive, which is one of the slowest components on a computer. This has been addressed with the introduction of solid state drives (SSD), but SSDs are still much more expensive than the slower and more widespread hard disk drive (HDD) technology.

4.2 General techniques for handling large volumes of data

Never-ending algorithms, out-of-memory errors, and speed issues are the most common challenges you face when working with large data. In this section, we'll investigate solutions to overcome or alleviate these problems.

The solutions can be divided into three categories: using the correct algorithms, choosing the right data structure, and using the right tools (figure 4.2).

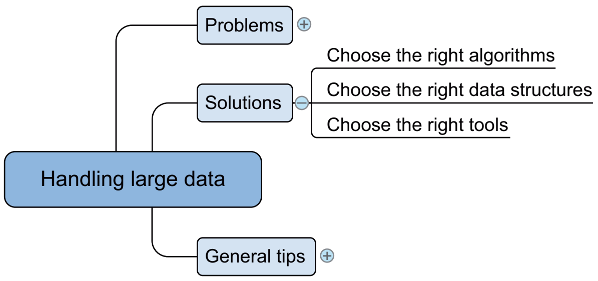


Figure 4.2 Overview of solutions for handling large data sets

No clear one-to-one mapping exists between the problems and solutions because many solutions address both lack of memory and computational performance. For instance, data set compression will help you solve memory issues because the data set becomes smaller. But this also affects computation speed with a shift from the slow hard disk to the fast CPU. Contrary to RAM (random access memory), the hard disc will store everything even after the power goes down, but writing to disc costs more time than changing information in the fleeting RAM. When constantly changing the information, RAM is thus preferable over the (more durable) hard disc. With an unpacked data set, numerous read and write operations (I/O) are occurring, but the CPU remains largely idle, whereas with the compressed data set the CPU gets its fair share of the workload. Keep this in mind while we explore a few solutions.

4.2.1 Choosing the right algorithm

Choosing the right algorithm can solve more problems than adding more or better hardware. An algorithm that's well suited for handling large data doesn't need to load the entire data set into memory to make predictions. Ideally, the algorithm also supports parallelized calculations. In this section we'll dig into three types of algorithms that can do that: *online algorithms*, *block algorithms*, and *MapReduce algorithms*, as shown in figure 4.3.

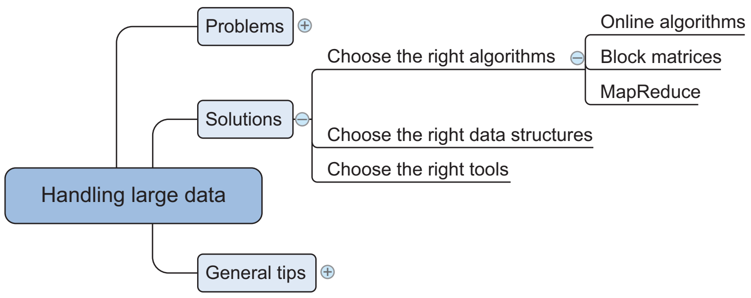


Figure 4.3 Overview of techniques to adapt algorithms to large data sets

ONLINE LEARNING ALGORITHMS

Several, but not all, machine learning algorithms can be trained using one observation at a time instead of taking all the data into memory. Upon the arrival of a new data point, the model is trained and the observation can be forgotten; its effect is now incorporated into the model's parameters. For example, a model used to predict the weather can use different parameters (like atmospheric pressure or temperature) in different regions. When the data from one region is loaded into the algorithm, it forgets about this raw data and moves on to the next region. This "use and forget" way of working is the perfect solution for the memory problem as a single observation is unlikely to ever be big enough to fill up all the memory of a modern-day computer.

Listing 4.1 shows how to apply this principle to a perceptron with online learning. A *perceptron* is one of the least complex machine learning algorithms used for binary classification (0 or 1); for instance, will the customer buy or not?

Listing 4.1 Training a perceptron by observation

The learning rate of an algorithm is the adjustment it makes every time a new observation comes in. If this is high, the model will adjust quickly to new observations but might "overshoot" and never get precise. An oversimplified example: the optimal (and unknown) weight for an x-variable = 0.75. Current estimation is 0.4 with a learning rate of 0.5; the adjustment = 0.5 (learning rate) * 1(size of error) * 1 (value of x) = 0.5. 0.4 (current weight) + 0.5 (adjustment) = 0.9 (new weight), instead of 0.75. The adjustment was too big to get the correct result.

Sets up perceptron class.

The __init__ method of any Python class is always run when creating an instance of the class. Several default values are set here.

```
import numpy as np
class perceptron():
    def __init__(self, X,y, threshold = 0.5,
learning_rate = 0.1, max_epochs = 10):
        self.threshold = threshold
        self.learning_rate = learning_rate
        self.X = X
        self.y = y
        self.max_epochs = max_epochs
```

The threshold is an arbitrary cutoff between 0 and 1 to decide whether the prediction becomes a 0 or a 1. Often it's 0.5, right in the middle, but it depends on the use case.

X and y variables are assigned to the class.

One epoch is one run through all the data. We allow for a maximum of 10 runs until we stop the perceptron.

Each observation will end up with a weight. The initialize function sets these weights for each incoming observation. We allow for 2 options: all weights start at 0 or they are assigned a small (between 0 and 0.05) random weight.

```
    def initialize(self, init_type = 'zeros'):
        if init_type == 'random':
            self.weights = np.random.rand(len(self.X[0])) * 0.05
        if init_type == 'zeros':
            self.weights = np.zeros(len(self.X[0]))
```

We start at the first epoch.

The training function.

True is always true, so technically this is a never-ending loop, but we build in several stop (break) conditions.

```
    def train(self):
        epoch = 0
        while True:
            error_count = 0
            epoch += 1
            for (X,y) in zip(self.X, self.y):
                error_count += self.train_observation(X,y,error_count)
```

Adds one to the current number of epochs.

Initiates the number of encountered errors at 0 for each epoch. This is important; if an epoch ends without errors, the algorithm converged and we're done.

We loop through the data and feed it to the train observation function, one observation at a time.

```
            if error_count == 0:
                print "training successful"
                break
            if epoch >= self.max_epochs:
                print "reached maximum epochs, no perfect prediction"
                break
```

If we reach the maximum number of allowed runs, we stop looking for a solution.

If by the end of the epoch we don't have an error, the training was successful.

The real value (y) is either 0 or 1; the prediction is also 0 or 1. If it's wrong we get an error of either 1 or -1.

The train observation function is run for every observation and will adjust the weights using the formula explained earlier.

```
    def train_observation(self,X,y, error_count):
        result = np.dot(X, self.weights) > self.threshold
        error = y - result
```

A prediction is made for this observation. Because it's binary, this will be either 0 or 1.

In case we have a wrong prediction (an error), we need to adjust the model.

Adds 1 to the error count.

For every predictor variable in the input vector (X), we'll adjust its weight.

```
        if error != 0:
            error_count += 1
            for index, value in enumerate(X):
                self.weights[index] +=  self.learning_rate * error * value
        return error_count
```

We return the error count because we need to evaluate it at the end of the epoch.

Adjusts the weight for every predictor variable using the learning rate, the error, and the actual value of the predictor variable.

```
    def predict(self, X):
        return int(np.dot(X, self.weights) > self.threshold)
```

The predict class.

The values of the predictor values are multiplied by their respective weights (this multiplication is done by np.dot). Then the outcome is compared to the overall threshold (here this is 0.5) to see if a 0 or 1 should be predicted.

```
X = [(1,0,0),(1,1,0),(1,1,1),(1,1,1),(1,0,1),(1,0,1)]
y = [1,1,0,0,1,1]

p = perceptron(X,y)
p.initialize()
p.train()
print p.predict((1,1,1))
print p.predict((1,0,1))
```

Our X (predictors) data matrix.

Our y (target) data vector.

We instantiate our perceptron class with the data from matrix X and vector y.

The weights for the predictors are initialized (as explained previously).

The perceptron model is trained. It will try to train until it either converges (no more errors) or it runs out of training runs (epochs).

We check what the perceptron would now predict given different values for the predictor variables. In the first case it will predict 0; in the second it predicts a 1.

We'll zoom in on parts of the code that might not be so evident to grasp without further explanation. We'll start by explaining how the `train_observation()` function works. This function has two large parts. The first is to calculate the prediction of an observation and compare it to the actual value. The second part is to change the weights if the prediction seems to be wrong.

The real value (y) is either 0 or 1; the prediction is also 0 or 1. If it's wrong we get an error of either 1 or -1.

The train observation function is run for every observation and will adjust the weights using the formula explained earlier.

A prediction is made for this observation. Because it's binary, this will be either 0 or 1.

```
def train_observation(self,X,y, error_count):
    result = np.dot(X, self.weights) > self.threshold
    error = y - result
    if error != 0:
        error_count += 1
        for index, value in enumerate(X):
            self.weights[index]+=self.learning_rate * error * value
    return error_count
```

In case we have a wrong prediction (an error), we need to adjust the model.

Adds 1 to error count.

Adjusts the weight for every predictor variable using the learning rate, the error, and the actual value of the predictor variable.

We return the error count because we need to evaluate it at the end of the epoch.

For every predictor variable in the input vector (X), we'll adjust its weight.

The prediction (y) is calculated by multiplying the input vector of independent variables with their respective weights and summing up the terms (as in linear regression). Then this value is compared with the threshold. If it's larger than the threshold, the algorithm will give a 1 as output, and if it's less than the threshold, the algorithm gives 0 as output. Setting the threshold is a subjective thing and depends on your business case. Let's say you're predicting whether someone has a certain lethal disease, with 1 being positive and 0 negative. In this case it's better to have a lower threshold: it's not as bad to be found positive and do a second investigation than it is to overlook the disease and let the patient die. The error is calculated, which will give the direction to the change of the weights.

```
result = np.dot(X, self.weights) > self.threshold
error = y - result
```

The weights are changed according to the sign of the error. The update is done with the learning rule for perceptrons. For every weight in the weight vector, you update its value with the following rule:

$$\Delta w_i = \alpha \varepsilon x_i$$

where Δw_i is the amount that the weight needs to be changed, α is the learning rate, ε is the error, and x_i is the ith value in the input vector (the ith predictor variable). The error count is a variable to keep track of how many observations are wrongly predicted in this epoch and is returned to the calling function. You add one observation to the error counter if the original prediction was wrong. An *epoch* is a single training run through all the observations.

```
if error != 0:
       error_count += 1
            for index, value in enumerate(X):
                self.weights[index] +=  self.learning_rate * error * value
```

The second function that we'll discuss in more detail is the `train()` function. This function has an internal loop that keeps on training the perceptron until it can either predict perfectly or until it has reached a certain number of training rounds (epochs), as shown in the following listing.

Listing 4.2 Using `train` functions

```
def train(self):
        epoch = 0
        while True:
            error_count = 0
            epoch += 1
            for (X,y) in zip(self.X, self.y):
                error_count += self.train_observation(X,y,error_count)
            if error_count == 0:
                print "training succesfull"
                break
            if epoch >= self.max_epochs:
                print "reached maximum epochs, no perfect prediction"
                break an loop to train the data
```

Training function.

Starts at the first epoch.

True is always true so technically this is a never-ending loop, but we build in several stop (break) conditions.

Initiates the number of encountered errors at 0 for each epoch. This is important, because if an epoch ends without errors, the algorithm converged and we're done.

Adds one to the current number of epochs.

We loop through the data and feed it to the train observation function, one observation at a time.

If by the end of the epoch we don't have an error, training was successful.

If we reach the maximum number of allowed runs, we stop looking for a solution.

Most online algorithms can also handle mini-batches; this way, you can feed them batches of 10 to 1,000 observations at once while using a sliding window to go over your data. You have three options:

- *Full batch learning (also called statistical learning)*—Feed the algorithm all the data at once. This is what we did in chapter 3.
- *Mini-batch learning*—Feed the algorithm a spoonful (100, 1000, …, depending on what your hardware can handle) of observations at a time.
- *Online learning*—Feed the algorithm one observation at a time.

Online learning techniques are related to *streaming algorithms*, where you see every data point only once. Think about incoming Twitter data: it gets loaded into the algorithms, and then the observation (tweet) is discarded because the sheer number of incoming tweets of data might soon overwhelm the hardware. Online learning algorithms differ from streaming algorithms in that they can see the same observations multiple times. True, the online learning algorithms and streaming algorithms can *both* learn from observations one by one. Where they differ is that *online algorithms* are also used on a static data source as well as on a streaming data source by presenting the data in small batches (as small as a single observation), which enables you to go over the data multiple times. This isn't the case with a *streaming algorithm*, where data flows into the system and you need to do the calculations typically immediately. They're similar in that they handle only a few at a time.

DIVIDING A LARGE MATRIX INTO MANY SMALL ONES

Whereas in the previous chapter we barely needed to deal with how exactly the algorithm estimates parameters, diving into this might sometimes help. By cutting a large data table into small matrices, for instance, we can still do a linear regression. The logic behind this matrix splitting and how a linear regression can be calculated with matrices can be found in the sidebar. It suffices to know for now that the Python libraries we're about to use will take care of the matrix splitting, and linear regression variable weights can be calculated using matrix calculus.

Block matrices and matrix formula of linear regression coefficient estimation

Certain algorithms can be translated into algorithms that use blocks of matrices instead of full matrices. When you partition a matrix into a block matrix, you divide the full matrix into parts and work with the smaller parts instead of the full matrix. In this case you can load smaller matrices into memory and perform calculations, thereby avoiding an out-of-memory error. Figure 4.4 shows how you can rewrite matrix addition A + B into submatrices.

$$A + B = \left[\begin{array}{ccc} a_{1,1} & \cdots & a_{1,m} \\ \hline \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{array}\right] + \left[\begin{array}{ccc} b_{1,1} & \cdots & b_{1,m} \\ \hline \vdots & \ddots & \vdots \\ b_{n,1} & \cdots & b_{n,m} \end{array}\right]$$

$$= \left[\begin{array}{ccc} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{j,1} & \cdots & a_{j,m} \\ \hline a_{j+1,1} & \cdots & a_{j+1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{array}\right] + \left[\begin{array}{ccc} b_{1,1} & \cdots & b_{1,m} \\ \vdots & \ddots & \vdots \\ b_{j,1} & \cdots & b_{j,m} \\ \hline b_{j+1,1} & \cdots & b_{j+1,m} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \cdots & b_{n,m} \end{array}\right] = \left[\begin{array}{c} A_1 \\ \hline A_2 \end{array}\right] + \left[\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right]$$

Figure 4.4 Block matrices can be used to calculate the sum of the matrices A and B.

(continued)

The formula in figure 4.4 shows that there's no difference between adding matrices A and B together in one step or first adding the upper half of the matrices and then adding the lower half.

All the common matrix and vector operations, such as multiplication, inversion, and singular value decomposition (a variable reduction technique like PCA), can be written in terms of block matrices.[1] Block matrix operations save memory by splitting the problem into smaller blocks and are easy to parallelize.

Although most numerical packages have highly optimized code, they work only with matrices that can fit into memory and will use block matrices in memory when advantageous. With out-of-memory matrices, they don't optimize this for you and it's up to you to partition the matrix into smaller matrices and to implement the block matrix version.

A *linear regression* is a way to predict continuous variables with a linear combination of its predictors; one of the most basic ways to perform the calculations is with a technique called *ordinary least squares*. The formula in matrix form is

$$\beta = (X^T X)^{-1} X^T y$$

where β is the coefficients you want to retrieve, X is the predictors, and y is the target variable.

The Python tools we have at our disposal to accomplish our task are the following:

- *bcolz* is a Python library that can store data arrays compactly and uses the hard drive when the array no longer fits into the main memory.
- *Dask* is a library that enables you to optimize the flow of calculations and makes performing calculations in parallel easier. It doesn't come packaged with the default Anaconda setup so make sure to use `conda install dask` on your virtual environment before running the code below. Note: some errors have been reported on importing Dask when using 64bit Python. Dask is dependent on a few other libraries (such as toolz), but the dependencies should be taken care of automatically by pip or conda.

The following listing demonstrates block matrix calculations with these libraries.

[1] For those who want to give it a try, Given transformations are easier to achieve than Householder transformations when calculating singular value decompositions.

Listing 4.3 Block matrix calculations with bcolz and Dask libraries

Number of observations (scientific notation). 1e4 = 10.000. Feel free to change this.

Creates fake data: np.arange(n).reshape(n/2,2) creates a matrix of 5000 by 2 (because we set n to 10.000). bc.carray = numpy is an array extension that can swap to disc. This is also stored in a compressed way. rootdir = 'ar.bcolz' --> creates a file on disc in case out of RAM. You can check this on your file system next to this ipython file or whatever location you ran this code from. mode = 'w' --> is the write mode. dtype = 'float64' --> is the storage type of the data (which is float numbers).

```
import dask.array as da
import bcolz as bc
import numpy as np
import dask

n = 1e4

ar = bc.carray(np.arange(n).reshape(n/2,2)  , dtype='float64',
     rootdir = 'ar.bcolz', mode = 'w')
y  = bc.carray(np.arange(n/2), dtype='float64', rootdir =
     'yy.bcolz', mode = 'w')


dax = da.from_array(ar, chunks=(5,5))
dy = da.from_array(y,chunks=(5,5))
```

Block matrices are created for the predictor variables (ar) and target (y). A block matrix is a matrix cut in pieces (blocks). da.from_array() reads data from disc or RAM (wherever it resides currently). chunks=(5,5): every block is a 5x5 matrix (unless < 5 observations or variables are left).

The XTX is defined (defining it as “lazy”) as the X matrix multiplied with its transposed version. This is a building block of the formula to do linear regression using matrix calculation.

Xy is the y vector multiplied with the transposed X matrix. Again the matrix is only defined, not calculated yet. This is also a building block of the formula to do linear regression using matrix calculation (see formula).

```
XTX = dax.T.dot(dax)
Xy  = dax.T.dot(dy)

coefficients = np.linalg.inv(XTX.compute()).dot(Xy.compute())

coef = da.from_array(coefficients,chunks=(5,5))
```

The coefficients are also put into a block matrix. We got a numpy array back from the last step so we need to explicitly convert it back to a “da array.”

```
ar.flush()
y.flush()
```

Flush memory data. It’s no longer needed to have large matrices in memory.

```
predictions = dax.dot(coef).compute()
print predictions
```

Score the model (make predictions).

The coefficients are calculated using the matrix linear regression function. np.linalg.inv() is the ^(-1) in this function, or “inversion” of the matrix. X.dot(y) --> multiplies the matrix X with another matrix y.

Note that you don’t need to use a block matrix inversion because XTX is a square matrix with size nr. of predictors * nr. of predictors. This is fortunate because Dask doesn't yet support block matrix inversion. You can find more general information on matrix arithmetic on the Wikipedia page at https://en.wikipedia.org/wiki/Matrix_(mathematics).

MAPREDUCE

MapReduce algorithms are easy to understand with an analogy: Imagine that you were asked to count all the votes for the national elections. Your country has 25 parties, 1,500 voting offices, and 2 million people. You could choose to gather all the voting tickets from every office individually and count them centrally, or you could ask the local offices to count the votes for the 25 parties and hand over the results to you, and you could then aggregate them by party.

Map reducers follow a similar process to the second way of working. They first map values to a key and then do an aggregation on that key during the reduce phase. Have a look at the following listing's pseudo code to get a better feeling for this.

Listing 4.4 MapReduce pseudo code example

```
For each person in voting office:
    Yield (voted_party, 1)
For each vote in voting office:
    add_vote_to_party()
```

One of the advantages of MapReduce algorithms is that they're easy to parallelize and distribute. This explains their success in distributed environments such as Hadoop, but they can also be used on individual computers. We'll take a more in-depth look at them in the next chapter, and an example (JavaScript) implementation is also provided in chapter 9. When implementing MapReduce in Python, you don't need to start from scratch. A number of libraries have done most of the work for you, such as Hadoopy, Octopy, Disco, or Dumbo.

4.2.2 *Choosing the right data structure*

Algorithms can make or break your program, but the way you store your data is of equal importance. Data structures have different storage requirements, but also influence the performance of *CRUD* (create, read, update, and delete) and other operations on the data set.

Figure 4.5 shows you have many different data structures to choose from, three of which we'll discuss here: sparse data, tree data, and hash data. Let's first have a look at sparse data sets.

SPARSE DATA

A sparse data set contains relatively little information compared to its entries (observations). Look at figure 4.6: almost everything is "0" with just a single "1" present in the second observation on variable 9.

Data like this might look ridiculous, but this is often what you get when converting textual data to binary data. Imagine a set of 100,000 completely unrelated Twitter

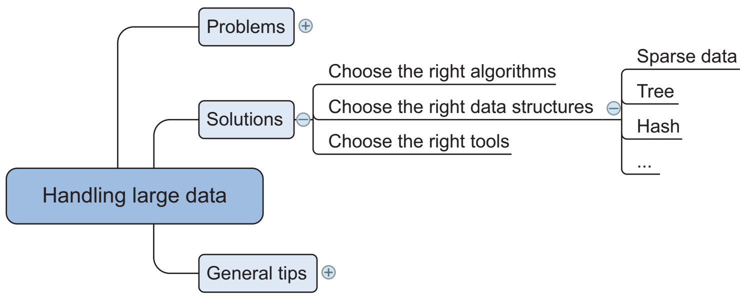
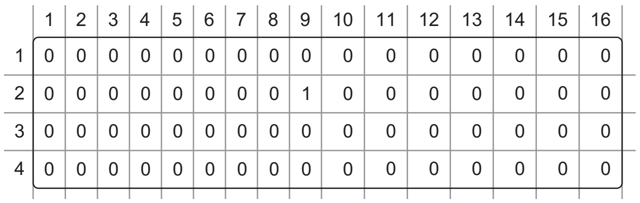


Figure 4.5 Overview of data structures often applied in data science when working with large data



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.6 Example of a sparse matrix: almost everything is 0; other values are the exception in a sparse matrix

tweets. Most of them probably have fewer than 30 words, but together they might have hundreds or thousands of distinct words. In the chapter on text mining we'll go through the process of cutting text documents into words and storing them as vectors. But for now imagine what you'd get if every word was converted to a binary variable, with "1" representing "present in this tweet," and "0" meaning "not present in this tweet." This would result in sparse data indeed. The resulting large matrix can cause memory problems even though it contains little information.

Luckily, data like this can be stored compacted. In the case of figure 4.6 it could look like this:

```
data = [(2,9,1)]
```

Row 2, column 9 holds the value 1.

Support for working with sparse matrices is growing in Python. Many algorithms now support or return sparse matrices.

TREE STRUCTURES

Trees are a class of data structure that allows you to retrieve information much faster than scanning through a table. A tree always has a root value and subtrees of children, each with its children, and so on. Simple examples would be your own family tree or a biological tree and the way it splits into branches, twigs, and leaves. Simple decision rules make it easy to find the child tree in which your data resides. Look at figure 4.7 to see how a tree structure enables you to get to the relevant information quickly.

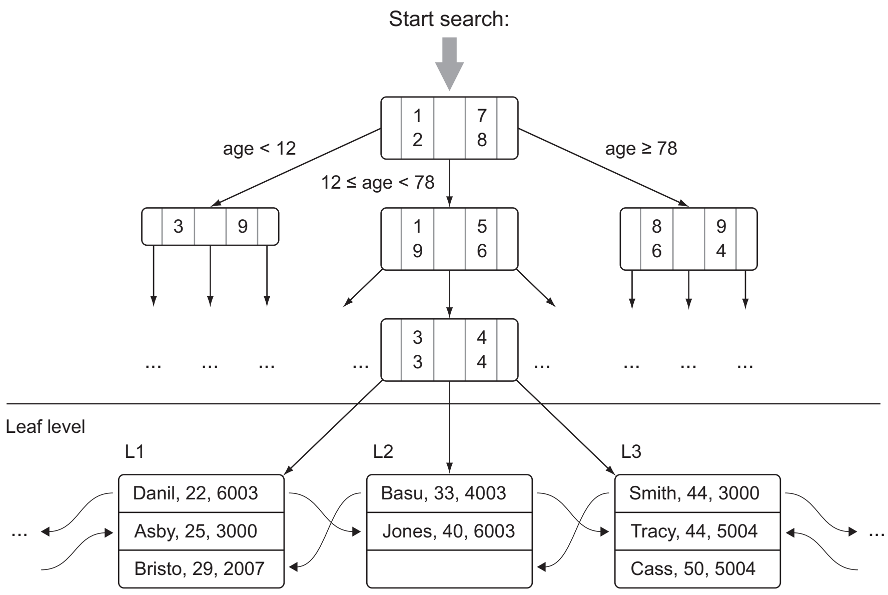


Figure 4.7 Example of a tree data structure: decision rules such as age categories can be used to quickly locate a person in a family tree

In figure 4.7 you start your search at the top and first choose an age category, because apparently that's the factor that cuts away the most alternatives. This goes on and on until you get what you're looking for. For whoever isn't acquainted with the Akinator, we recommend visiting http://en.akinator.com/. The Akinator is a djinn in a magical lamp that tries to guess a person in your mind by asking you a few questions about him or her. Try it out and be amazed . . . or see how this magic is a tree search.

Trees are also popular in databases. Databases prefer not to scan the table from the first line until the last, but to use a device called an *index* to avoid this. Indices are often based on data structures such as trees and hash tables to find observations faster. The use of an index speeds up the process of finding data enormously. Let's look at these hash tables.

HASH TABLES

Hash tables are data structures that calculate a key for every value in your data and put the keys in a bucket. This way you can quickly retrieve the information by looking in the right bucket when you encounter the data. Dictionaries in Python are a hash table implementation, and they're a close relative of key-value stores. You'll encounter them in the last example of this chapter when you build a recommender system within a database. Hash tables are used extensively in databases as indices for fast information retrieval.

4.2.3 Selecting the right tools

With the right class of algorithms and data structures in place, it's time to choose the right tool for the job. The right tool can be a Python library or at least a tool that's controlled from Python, as shown figure 4.8. The number of helpful tools available is enormous, so we'll look at only a handful of them.

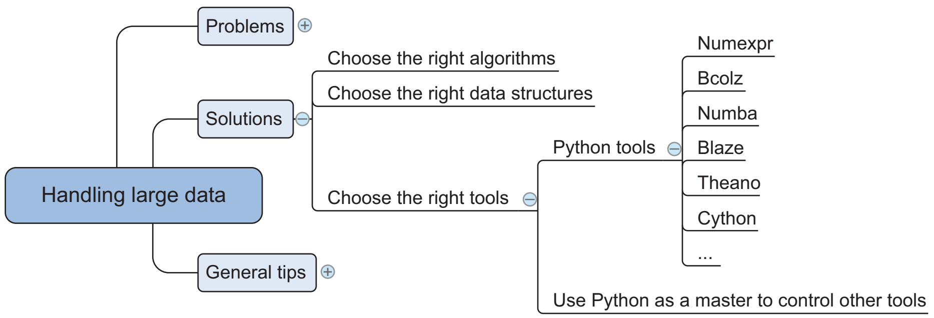


Figure 4.8 Overview of tools that can be used when working with large data

Python tools

Python has a number of libraries that can help you deal with large data. They range from smarter data structures over code optimizers to just-in-time compilers. The following is a list of libraries we like to use when confronted with large data:

- *Cython*—The closer you get to the actual hardware of a computer, the more vital it is for the computer to know what types of data it has to process. For a computer, adding 1 + 1 is different from adding 1.00 + 1.00. The first example consists of integers and the second consists of floats, and these calculations are performed by different parts of the CPU. In Python you don't have to specify what data types you're using, so the Python compiler has to infer them. But inferring data types is a slow operation and is partially why Python isn't one of the fastest languages available. Cython, a superset of Python, solves this problem by forcing the programmer to specify the data type while developing the program. Once the compiler has this information, it runs programs much faster. See http://cython.org/ for more information on Cython.
- *Numexpr*—Numexpr is at the core of many of the big data packages, as is NumPy for in-memory packages. Numexpr is a numerical expression evaluator for NumPy but can be many times faster than the original NumPy. To achieve

this, it rewrites your expression and uses an internal (just-in-time) compiler. See https://github.com/pydata/numexpr for details on Numexpr.

- *Numba*—Numba helps you to achieve greater speed by compiling your code right before you execute it, also known as *just-in-time compiling*. This gives you the advantage of writing high-level code but achieving speeds similar to those of C code. Using Numba is straightforward; see http://numba.pydata.org/.
- *Bcolz*—Bcolz helps you overcome the out-of-memory problem that can occur when using NumPy. It can store and work with arrays in an optimal compressed form. It not only slims down your data need but also uses Numexpr in the background to reduce the calculations needed when performing calculations with bcolz arrays. See http://bcolz.blosc.org/.
- *Blaze*—Blaze is ideal if you want to use the power of a database backend but like the "Pythonic way" of working with data. Blaze will translate your Python code into SQL but can handle many more data stores than relational databases such as CSV, Spark, and others. Blaze delivers a unified way of working with many databases and data libraries. Blaze is still in development, though, so many features aren't implemented yet. See http://blaze.readthedocs.org/en/latest/index.html.
- *Theano*—Theano enables you to work directly with the graphical processing unit (GPU) and do symbolical simplifications whenever possible, and it comes with an excellent just-in-time compiler. On top of that it's a great library for dealing with an advanced but useful mathematical concept: tensors. See http://deeplearning.net/software/theano/.
- *Dask*—Dask enables you to optimize your flow of calculations and execute them efficiently. It also enables you to distribute calculations. See http://dask.pydata.org/en/latest/.

These libraries are mostly about using Python itself for data processing (apart from Blaze, which also connects to databases). To achieve high-end performance, you can use Python to communicate with all sorts of databases or other software.

USE PYTHON AS A MASTER TO CONTROL OTHER TOOLS

Most software and tool producers support a Python interface to their software. This enables you to tap into specialized pieces of software with the ease and productivity that comes with Python. This way Python sets itself apart from other popular data science languages such as R and SAS. You should take advantage of this luxury and exploit the power of specialized tools to the fullest extent possible. Chapter 6 features a case study using Python to connect to a NoSQL database, as does chapter 7 with graph data.

Let's now have a look at more general helpful tips when dealing with large data.

4.3 General programming tips for dealing with large data sets

The tricks that work in a general programming context still apply for data science. Several might be worded slightly differently, but the principles are essentially the same for all programmers. This section recapitulates those tricks that are important in a data science context.

You can divide the general tricks into three parts, as shown in the figure 4.9 mind map:

- *Don't reinvent the wheel.* Use tools and libraries developed by others.
- *Get the most out of your hardware.* Your machine is never used to its full potential; with simple adaptions you can make it work harder.
- *Reduce the computing need.* Slim down your memory and processing needs as much as possible.

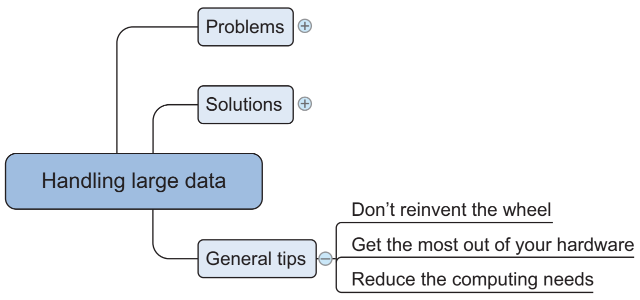


Figure 4.9 Overview of general programming best practices when working with large data

"Don't reinvent the wheel" is easier said than done when confronted with a specific problem, but your first thought should always be, 'Somebody else must have encountered this same problem before me.'

4.3.1 Don't reinvent the wheel

"Don't repeat anyone" is probably even better than "don't repeat yourself." Add value with your actions: make sure that they matter. Solving a problem that has already been solved is a waste of time. As a data scientist, you have two large rules that can help you deal with large data and make you much more productive, to boot:

- *Exploit the power of databases.* The first reaction most data scientists have when working with large data sets is to prepare their analytical base tables inside a database. This method works well when the features you want to prepare are fairly simple. When this preparation involves advanced modeling, find out if it's possible to employ user-defined functions and procedures. The last example of this chapter is on integrating a database into your workflow.
- *Use optimized libraries.* Creating libraries like Mahout, Weka, and other machine-learning algorithms requires time and knowledge. They are highly optimized

and incorporate best practices and state-of-the art technologies. Spend your time on getting things done, not on reinventing and repeating others people's efforts, unless it's for the sake of understanding how things work.

Then you must consider your hardware limitation.

4.3.2 Get the most out of your hardware

Resources on a computer can be idle, whereas other resources are over-utilized. This slows down programs and can even make them fail. Sometimes it's possible (and necessary) to shift the workload from an overtaxed resource to an underutilized resource using the following techniques:

- *Feed the CPU compressed data.* A simple trick to avoid CPU starvation is to feed the CPU compressed data instead of the inflated (raw) data. This will shift more work from the hard disk to the CPU, which is exactly what you want to do, because a hard disk can't follow the CPU in most modern computer architectures.
- *Make use of the GPU.* Sometimes your CPU and not your memory is the bottleneck. If your computations are parallelizable, you can benefit from switching to the GPU. This has a much higher throughput for computations than a CPU. The GPU is enormously efficient in parallelizable jobs but has less cache than the CPU. But it's pointless to switch to the GPU when your hard disk is the problem. Several Python packages, such as Theano and NumbaPro, will use the GPU without much programming effort. If this doesn't suffice, you can use a CUDA (Compute Unified Device Architecture) package such as PyCUDA. It's also a well-known trick in bitcoin mining, if you're interested in creating your own money.
- *Use multiple threads.* It's still possible to parallelize computations on your CPU. You can achieve this with normal Python threads.

4.3.3 Reduce your computing needs

"Working smart + hard = achievement." This also applies to the programs you write. The best way to avoid having large data problems is by removing as much of the work as possible up front and letting the computer work only on the part that can't be skipped. The following list contains methods to help you achieve this:

- *Profile your code and remediate slow pieces of code.* Not every piece of your code needs to be optimized; use a profiler to detect slow parts inside your program and remediate these parts.
- *Use compiled code whenever possible, certainly when loops are involved.* Whenever possible use functions from packages that are optimized for numerical computations instead of implementing everything yourself. The code in these packages is often highly optimized and compiled.
- *Otherwise, compile the code yourself.* If you can't use an existing package, use either a just-in-time compiler or implement the slowest parts of your code in a

lower-level language such as C or Fortran and integrate this with your codebase. If you make the step to *lower-level languages* (languages that are closer to the universal computer bytecode), learn to work with computational libraries such as LAPACK, BLAST, Intel MKL, and ATLAS. These are highly optimized, and it's difficult to achieve similar performance to them.

- *Avoid pulling data into memory.* When you work with data that doesn't fit in your memory, avoid pulling everything into memory. A simple way of doing this is by reading data in chunks and parsing the data on the fly. This won't work on every algorithm but enables calculations on extremely large data sets.
- *Use generators to avoid intermediate data storage.* Generators help you return data per observation instead of in batches. This way you avoid storing intermediate results.
- *Use as little data as possible.* If no large-scale algorithm is available and you aren't willing to implement such a technique yourself, then you can still train your data on only a sample of the original data.
- *Use your math skills to simplify calculations as much as possible.* Take the following equation, for example: $(a + b)^2 = a^2 + 2ab + b^2$. The left side will be computed much faster than the right side of the equation; even for this trivial example, it could make a difference when talking about big chunks of data.

4.4 Case study 1: Predicting malicious URLs

The internet is probably one of the greatest inventions of modern times. It has boosted humanity's development, but not everyone uses this great invention with honorable intentions. Many companies (Google, for one) try to protect us from fraud by detecting malicious websites for us. Doing so is no easy task, because the internet has billions of web pages to scan. In this case study we'll show how to work with a data set that no longer fits in memory.

What we'll use

- *Data*—The data in this case study was made available as part of a research project. The project contains data from 120 days, and each observation has approximately 3,200,000 features. The target variable contains 1 if it's a malicious website and -1 otherwise. For more information, please see "Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs."[2]
- *The Scikit-learn library*—You should have this library installed in your Python environment at this point, because we used it in the previous chapter.

As you can see, we won't be needing much for this case, so let's dive into it.

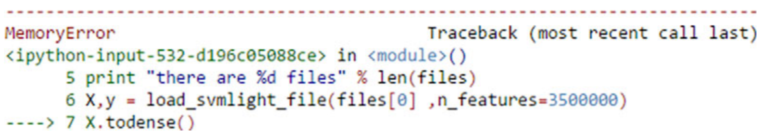
[2] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker, "Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs," Proceedings of the ACM SIGKDD Conference, Paris (June 2009), 1245–53.

4.4.1 Step 1: Defining the research goal

The goal of our project is to detect whether certain URLs can be trusted or not. Because the data is so large we aim to do this in a memory-friendly way. In the next step we'll first look at what happens if we don't concern ourselves with memory (RAM) issues.

4.4.2 Step 2: Acquiring the URL data

Start by downloading the data from http://sysnet.ucsd.edu/projects/url/#datasets and place it in a folder. Choose the data in SVMLight format. SVMLight is a text-based format with one observation per row. To save space, it leaves out the zeros.



```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-532-d196c05088ce> in <module>()
      5 print "there are %d files" % len(files)
      6 X,y = load_svmlight_file(files[0] ,n_features=3500000)
----> 7 X.todense()
```

Figure 4.10 Memory error when trying to take a large data set into memory

The following listing and figure 4.10 show what happens when you try to read in 1 file out of the 120 and create the normal matrix as most algorithms expect. The `todense()` method changes the data from a special file format to a normal matrix where every entry contains a value.

Listing 4.5 Generating an out-of-memory error

```
import glob
from sklearn.datasets import load_svmlight_file
files = glob.glob('C:\Users\Gebruiker\Downloads\
url_svmlight.tar\url_svmlight\*.svm')
files = glob.glob('C:\Users\Gebruiker\Downloads\
url_svmlight\url_svmlight\*.svm')
print "there are %d files" % len(files)
X,y = load_svmlight_file(files[0],n_features=3231952)
X.todense()
```

Points to files (Linux).

Points to files (Windows: tar file needs to be untarred first).

Indication of number of files.

Loads files.

The data is a big, but sparse, matrix. By turning it into a dense matrix (every 0 is represented in the file), we create an out-of-memory error.

Surprise, surprise, we get an out-of-memory error. That is, unless you run this code on a huge machine. After a few tricks you'll no longer run into these memory problems and will detect 97% of the malicious sites.

Tools and techniques

We ran into a memory error while loading a single file—still 119 to go. Luckily, we have a few tricks up our sleeve. Let's try these techniques over the course of the case study:

- Use a sparse representation of data.
- Feed the algorithm compressed data instead of raw data.
- Use an online algorithm to make predictions.

We'll go deeper into each "trick" when we get to use it. Now that we have our data locally, let's access it. Step 3 of the data science process, data preparation and cleansing, isn't necessary in this case because the URLs come pre-cleaned. We'll need a form of exploration before unleashing our learning algorithm, though.

4.4.3 Step 4: Data exploration

To see if we can even apply our first trick (sparse representation), we need to find out whether the data does indeed contain lots of zeros. We can check this with the following piece of code:

```
print "number of non-zero entries %2.6f"  % float((X.nnz)/(float(X.shape[0])
* float(X.shape[1])))
```

This outputs the following:

```
number of non-zero entries 0.000033
```

Data that contains little information compared to zeros is called *sparse data*. This can be saved more compactly if you store the data as `[(0,0,1),(4,4,1)]` instead of

```
[[1,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,1]]
```

One of the file formats that implements this is SVMLight, and that's exactly why we downloaded the data in this format. We're not finished yet, though, because we need to get a feel of the dimensions within the data.

To get this information we already need to keep the data compressed while checking for the maximum number of observations and variables. We also need to *read in data file by file*. This way you consume even less memory. A second trick is to feed the CPU compressed files. In our example, it's already packed in the tar.gz format. You unpack a file only when you need it, without writing it to the hard disk (the slowest part of your computer).

For our example, shown in listing 4.6, we'll only work on the first 5 files, but feel free to use all of them.

Listing 4.6 Checking data size

```
import tarfile
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import classification_report
from sklearn.datasets import load_svmlight_file
import numpy as np

uri = 'D:\Python Book\Chapter 4\url_svmlight.tar.gz'
tar = tarfile.open(uri,"r:gz")
max_obs = 0
max_vars = 0
i = 0
split = 5
for tarinfo in tar:
    print " extracting %s,f size %s" % (tarinfo.name, tarinfo.size)
    if tarinfo.isfile():
        f = tar.extractfile(tarinfo.name)
        X,y = load_svmlight_file(f)
        max_vars = np.maximum(max_vars, X.shape[0])
        max_obs = np.maximum(max_obs, X.shape[1])

    if i  > split:
        break
    i+= 1

print "max X = %s, max y dimension = %s" % (max_obs, max_vars )
```

We don't know how many features we have, so let's initialize it at 0.

We don't know how many observations we have, so let's initialize it at 0.

The uri variable holds the location in which you saved the downloaded files. You'll need to fill out this uri variable yourself for the code to run on your computer.

All files together take up around 2.05 Gb. The trick here is to leave the data compressed in main memory and only unpack what you need.

Stop at the 5th file (instead of all of them, for demonstration purposes).

Initialize file counter at 0.

Use a helper function, load_svmlight_file() to load a specific file.

We unpack the files one by one to reduce the memory needed.

Adjust maximum number of observations and variables when necessary (big file).

Stop when we reach 5 files.

Print results.

Part of the code needs some extra explanation. In this code we loop through the svm files inside the tar archive. We unpack the files one by one to reduce the memory needed. As these files are in the SVM format, we use a helper, `functionload_svmlight_file()` to load a specific file. Then we can see how many observations and variables the file has by checking the shape of the resulting data set.

Armed with this information we can move on to model building.

4.4.4 Step 5: Model building

Now that we're aware of the dimensions of our data, we can apply the same two tricks (sparse representation of compressed file) and add the third (using an online algorithm), in the following listing. Let's find those harmful websites!

Listing 4.7 Creating a model to distinguish the malicious from the normal URLs

We know number of features from data exploration.

The target variable can be 1 or -1. "1": website safe to visit, "-1": website unsafe.

Set up stochastic gradient classifier.

```
classes = [-1,1]
sgd = SGDClassifier(loss="log")
n_features=3231952
split = 5
i = 0
for tarinfo in tar:
    if i  > split:
        break
    if tarinfo.isfile():
        f = tar.extractfile(tarinfo.name)
        X,y = load_svmlight_file(f,n_features=n_features)
        if i < split:
            sgd.partial_fit(X, y, classes=classes)
        if i == split:
            print classification_report(sgd.predict(X),y)
    i += 1
```

All files together take up around 2.05 Gb. The trick here is to leave data compressed in main memory and only unpack what you need.

We unpack the files one by one to reduce the memory needed.

Use a helper function, load_svmlight_file() to load a specific file.

Third important thing is online algorithm. It can be fed data points file by file (batches).

Stop when we reach 5 files and print results.

Initialize file counter at 0.

Stop at 5th file (instead of all of them, for demonstration purposes).

The code in the previous listing looks fairly similar to what we did before, apart from the stochastic gradient descent classifier `SGDClassifier()`.

Here, we trained the algorithm iteratively by presenting the observations in one file with the `partial_fit()` function.

Looping through only the first 5 files here gives the output shown in table 4.1. The table shows classification diagnostic measures: precision, recall, F1-score, and support.

Table 4.1 Classification problem: Can a website be trusted or not?

	precision	recall	f1-score	support
-1	0.97	0.99	0.98	14045
1	0.97	0.94	0.96	5955
avg/total	0.97	0.97	0.97	20000

Only 3% (1 - 0.97) of the malicious sites aren't detected (*precision*), and 6% (1 - 0.94) of the sites detected are falsely accused (*recall*). This is a decent result, so we can conclude that the methodology works. If we rerun the analysis, the result might be slightly different, because the algorithm could converge slightly differently. If you don't mind waiting a while, you can go for the full data set. You can now handle all the data without problems. We won't have a sixth step (presentation or automation) in this case study.

Now let's look at a second application of our techniques; this time you'll build a recommender system inside a database. For a well-known example of recommender systems visit the Amazon website. While browsing, you'll soon be confronted with recommendations: "People who bought this product also bought…"

4.5 Case study 2: Building a recommender system inside a database

In reality most of the data you work with is stored in a relational database, but most databases aren't suitable for data mining. But as shown in this example, it's possible to adapt our techniques so you can do a large part of the analysis inside the database itself, thereby profiting from the database's query optimizer, which will optimize the code for you. In this example we'll go into how to use the hash table data structure and how to use Python to control other tools.

4.5.1 Tools and techniques needed

Before going into the case study we need to have a quick look at the required tools and theoretical background to what we're about to do here.

TOOLS

- *MySQL database*—Needs a MySQL database to work with. If you haven't installed a MySQL community server, you can download one from www.mysql.com. Appendix C: "Installing a MySQL server" explains how to set it up.
- *MySQL database connection Python library*—To connect to this server from Python you'll also need to install SQLAlchemy or another library capable of communicating with MySQL. We're using MySQLdb. On Windows you can't use Conda right off the bat to install it. First install Binstar (another package management service) and look for the appropriate mysql-python package for your Python setup.

```
conda install binstar
binstar search -t conda mysql-python
```

The following command entered into the Windows command line worked for us (after activating the Python environment):

```
conda install --channel https://conda.binstar.org/krisvanneste mysql-python
```

Again, feel free to go for the SQLAlchemy library if that's something you're more comfortable with.

- We will also need the *pandas* python library, but that should already be installed by now.

With the infrastructure in place, let's dive into a few of the techniques.

Techniques

A simple recommender system will look for customers who've rented similar movies as you have and then suggest those that the others have watched but you haven't seen yet. This technique is called *k-nearest neighbors* in machine learning.

A customer who behaves similarly to you isn't necessarily *the* most similar customer. You'll use a technique to ensure that you can find similar customers (local optima) without guarantees that you've found the best customer (global optimum). A common technique used to solve this is called *Locality-Sensitive Hashing*. A good overview of papers on this topic can be found at http://www.mit.edu/~andoni/LSH/.

The idea behind Locality-Sensitive Hashing is simple: Construct functions that map similar customers close together (they're put in a bucket with the same label) and make sure that objects that are different are put in different buckets.

Central to this idea is a function that performs the mapping. This function is called a hash function: a function that maps any range of input to a fixed output. The simplest hash function concatenates the values from several random columns. It doesn't matter how many columns (scalable input); it brings it back to a single column (fixed output).

You'll set up three hash functions to find similar customers. The three functions take the values of three movies:

- The first function takes the values of movies 10, 15, and 28.
- The second function takes the values of movies 7, 18, and 22.
- The last function takes the values of movies 16, 19, and 30.

This will ensure that the customers who are in the same bucket share at least several movies. But the customers inside one bucket might still differ on the movies that weren't included in the hashing functions. To solve this you still need to compare the customers within the bucket with each other. For this you need to create a new distance measure.

The distance that you'll use to compare customers is called the hamming distance. The hamming distance is used to calculate how much two strings differ. The distance is defined as the number of different characters in a string. Table 4.2 offers a few examples of the hamming distance.

Table 4.2 Examples of calculating the hamming distance

String 1	String 2	Hamming distance
Hat	Cat	1
Hat	Mad	2
Tiger	Tigre	2
Paris	Rome	5

Comparing multiple columns is an expensive operation, so you'll need a trick to speed this up. Because the columns contain a binary (0 or 1) variable to indicate whether a customer has bought a movie or not, you can concatenate the information so that the same information is contained in a new column. Table 4.3 shows the "movies" variable that contains as much information as all the movie columns combined.

Table 4.3 Combining the information from different columns into the movies column. This is also how DNA works: all information in a long string.

Column 1	Movie 1	Movie 2	Movie 3	Movie 4	movies
Customer 1	1	0	1	1	1011
Customer 2	0	0	0	1	0001

This allows you to calculate the hamming distance much more efficiently. By handling this operator as a bit, you can exploit the XOR operator. The outcome of the XOR operator (^) is as follows:

```
1^1 = 0
1^0 = 1
0^1 = 1
0^0 = 0
```

With this in place, the process to find similar customers becomes very simple. Let's first look at it in pseudo code:

Preprocessing:

1. Define *p* (for instance, 3) functions that select *k* (for instance, 3) entries from the vector of movies. Here we take 3 functions (p) that each take 3 (k) movies.
2. Apply these functions to every point and store them in a separate column. (In literature each function is called a hash function and each column will store a bucket.)

Querying point q:

1. Apply the same p functions to the point (observation) q you want to query.
2. Retrieve for every function the points that correspond to the result in the corresponding bucket.

   Stop when you've retrieved all the points in the buckets or reached 2p points (for example 10 if you have 5 functions).
3. Calculate the distance for each point and return the points with the minimum distance.

Let's look at an actual implementation in Python to make this all clearer.

4.5.2 Step 1: Research question

Let's say you're working in a video store and the manager asks you if it's possible to use the information on what movies people rent to predict what other movies they might like. Your boss has stored the data in a MySQL database, and it's up to you to do the analysis. What he is referring to is a recommender system, an automated system that learns people's preferences and recommends movies and other products the customers haven't tried yet. The goal of our case study is to create a memory-friendly recommender system. We'll achieve this using a database and a few extra tricks. We're going to create the data ourselves for this case study so we can skip the data retrieval step and move right into data preparation. And after that we can skip the data exploration step and move straight into model building.

4.5.3 Step 3: Data preparation

The data your boss has collected is shown in table 4.4. We'll create this data ourselves for the sake of demonstration.

Table 4.4 Excerpt from the client database and the movies customers rented

Customer	Movie 1	Movie 2	Movie 3	...	Movie 32
Jack Dani	1	0	0		1
Wilhelmson	1	1	0		1
...					
Jane Dane	0	0	1		0
Xi Liu	0	0	0		1
Eros Mazo	1	1	0		1
...					

For each customer you get an indication of whether they've rented the movie before (1) or not (0). Let's see what else you'll need so you can give your boss the recommender system he desires.

First let's connect Python to MySQL to create our data. Make a connection to MySQL using your username and password. In the following listing we used a database called "test". Replace the user, password, and database name with the appropriate values for your setup and retrieve the connection and the cursor. A database cursor is a control structure that remembers where you are currently in the database.

Listing 4.8 Creating customers in the database

```
import MySQLdb
import pandas as pd

user = '****'
password = '****'
database = 'test'
mc = MySQLdb.connect('localhost',user,password,database)
cursor = mc.cursor()

nr_customers = 100
colnames = ["movie%d" %i for i in range(1,33)]
pd.np.random.seed(2015)
generated_customers = pd.np.random.randint(0,2,32 *
     nr_customers).reshape(nr_customers,32)

data = pd.DataFrame(generated_customers, columns = list(colnames))
data.to_sql('cust',mc, flavor = 'mysql', index = True, if_exists =
     'replace', index_label = 'cust_id')
```

First we establish the connection; you'll need to fill out your own username, password, and schema-name (variable "database").

Next we simulate a database with customers and create a few observations.

Store the data inside a Pandas data frame and write the data frame in a MySQL table called "cust". If this table already exists, replace it.

We create 100 customers and randomly assign whether they did or didn't see a certain movie, and we have 32 movies in total. The data is first created in a Pandas data frame but is then turned into SQL code. Note: You might run across a warning when running this code. The warning states: *The "mysql" flavor with DBAPI connection is deprecated and will be removed in future versions. MySQL will be further supported with SQLAlchemy engines.* Feel free to already switch to SQLAlchemy or another library. We'll use SQLAlchemy in other chapters, but used MySQLdb here to broaden the examples.

To efficiently query our database later on we'll need additional data preparation, including the following things:

- Creating bit strings. The bit strings are compressed versions of the columns' content (0 and 1 values). First these binary values are concatenated; then the resulting bit string is reinterpreted as a number. This might sound abstract now but will become clearer in the code.
- Defining hash functions. The hash functions will in fact create the bit strings.
- Adding an index to the table, to quicken data retrieval.

CREATING BIT STRINGS

Now you make an intermediate table suited for querying, apply the hash functions, and represent the sequence of bits as a decimal number. Finally, you can place them in a table.

First, you need to create bit strings. You need to convert the string "11111111" to a binary or a numeric value to make the hamming function work. We opted for a numeric representation, as shown in the next listing.

Listing 4.9 Creating bit strings

We represent the string as a numeric value. The string will be a concatenation of zeros (0) and ones (1) because these indicate whether someone has seen a certain movie or not. The strings are then regarded as bit code. For example: 0011 is the same as the number 3. What def createNum() does: takes in 8 values, concatenates these 8 column values and turns them into a string, then turns the byte code of the string into a number.

```
def createNum(x1,x2,x3,x4,x5,x6,x7,x8):
    return  [int('%d%d%d%d%d%d%d%d' % (i1,i2,i3,i4,i5,i6,i7,i8),2)
for (i1,i2,i3,i4,i5,i6,i7,i8) in zip(x1,x2,x3,x4,x5,x6,x7,x8)]

assert int('1111',2) == 15
assert int('1100',2) == 12
assert createNum([1,1],[1,1],[1,1],[1,1],[1,1],[1,1],[1,0],[1,0])
     == [255,252]

store = pd.DataFrame()
store['bit1'] = createNum(data.movie1,
     data.movie2,data.movie3,data.movie4,data.movie5,
data.movie6,data.movie7,data.movie8)
store['bit2'] = createNum(data.movie9,
     data.movie10,data.movie11,data.movie12,data.movie13,
data.movie14,data.movie15,data.movie16)
store['bit3'] = createNum(data.movie17,
     data.movie18,data.movie19,data.movie20,data.movie21,
data.movie22,data.movie23,data.movie24)
store['bit4'] = createNum(data.movie25,
     data.movie26,data.movie27,data.movie28,data.movie29,
data.movie30,data.movie31,data.movie32)
```

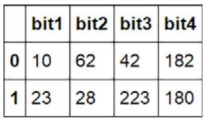
Translate the movie column to 4 bit strings in numeric form. Each bit string represents 8 movies. 4*8 = 32 movies. Note: you could use a 32-bit string instead of 4*8 to keep the code short.

Test if the function works correctly. Binary code 1111 is the same as 15 (=1*8+1*4+1*2+1*1). If the assert fails, it will raise an assert error; otherwise nothing will happen.

By converting the information of 32 columns into 4 numbers, we compressed it for later lookup. Figure 4.11 shows what we get when asking for the first 2 observations (customer movie view history) in this new format.

```
store[0:2]
```

The next step is to create the hash functions, because they'll enable us to sample the data we'll use to determine whether two customers have similar behavior.



	bit1	bit2	bit3	bit4
0	10	62	42	182
1	23	28	223	180

Figure 4.11 First 2 customers' information on all 32 movies after bit string to numeric conversion

CREATING A HASH FUNCTION

The hash functions we create take the values of movies for a customer. We decided in the theory part of this case study to create 3 hash functions: the first function combines the movies 10, 5, and 18; the second combines movies 7, 18, and 22; and the third one combines 16, 19, and 30. It's up to you if you want to pick others; this can be picked randomly. The following code listing shows how this is done.

Listing 4.10 Creating hash functions

Define hash function (it is exactly like the createNum() function without the final conversion to a number and for 3 columns instead of 8).

Test if it works correctly (if no error is raised, it works). It's sampling on columns but all observations will be selected.

```
def hash_fn(x1,x2,x3):
    return [b'%d%d%d' % (i,j,k) for (i,j,k) in zip(x1,x2,x3)]

assert hash_fn([1,0],[1,1],[0,0]) == [b'110',b'010']

store['bucket1'] = hash_fn(data.movie10, data.movie15,data.movie28)
store['bucket2'] = hash_fn(data.movie7, data.movie18,data.movie22)
store['bucket3'] = hash_fn(data.movie16, data.movie19,data.movie30)
store.to_sql('movie_comparison',mc, flavor = 'mysql', index = True,
             index_label = 'cust_id', if_exists = 'replace')
```

Store this information in database.

Create hash values from customer movies, respectively [10,15, 28], [7,18, 22], [16,19, 30].

The hash function concatenates the values from the different movies into a binary value like what happened before in the `createNum()` function, only this time we don't convert to numbers and we only take 3 movies instead of 8 as input. The assert function shows how it concatenates the 3 values for every observation. When the client has bought movie 10 but not movies 15 and 28, it will return b'100' for bucket 1. When the client bought movies 7 and 18, but not 22, it will return b'110' for bucket 2. If we look at the current result we see the 4 variables we created earlier (bit1, bit2, bit3, bit4) from the 9 handpicked movies (figure 4.12).

	bit1	bit2	bit3	bit4	bucket1	bucket2	bucket3
0	10	62	42	182	011	100	011
1	23	28	223	180	001	111	001

Figure 4.12 Information from the bit string compression and the 9 sampled movies

The last trick we'll apply is indexing the customer table so lookups happen more quickly.

Adding an index to the table

Now you must add indices to speed up retrieval as needed in a real-time system. This is shown in the next listing.

Listing 4.11 Creating an index

```
def createIndex(column, cursor):
    sql = 'CREATE INDEX %s ON movie_comparison (%s);' % (column, column)
    cursor.execute(sql)

createIndex('bucket1',cursor)
createIndex('bucket2',cursor)
createIndex('bucket3',cursor)
```

Create function to easily create indices. Indices will quicken retrieval.

Put index on bit buckets.

With the data indexed we can now move on to the "model building part." In this case study no actual machine learning or statistical model is implemented. Instead we'll use a far simpler technique: string distance calculation. Two strings can be compared using the hamming distance as explained earlier in the theoretical intro to the case study.

4.5.4 Step 5: Model building

To use the hamming distance in the database we need to define it as a function.

Creating the hamming distance function

We implement this as *a user-defined function*. This function can calculate the distance for a 32-bit integer (actually 4*8), as shown in the following listing.

Listing 4.12 Creating the hamming distance

```
Sql = '''
CREATE FUNCTION HAMMINGDISTANCE(
  A0 BIGINT, A1 BIGINT, A2 BIGINT, A3 BIGINT,
  B0 BIGINT, B1 BIGINT, B2 BIGINT, B3 BIGINT
)

RETURNS INT DETERMINISTIC
RETURN
  BIT_COUNT(A0 ^ B0) +
  BIT_COUNT(A1 ^ B1) +
  BIT_COUNT(A2 ^ B2) +
  BIT_COUNT(A3 ^ B3); '''

cursor.execute(Sql)

Sql = '''Select hammingdistance(
    b'11111111',b'00000000',b'11011111',b'11111111'
,b'11111111',b'10001001',b'11011111',b'11111111'
)'''
pd.read_sql(Sql,mc)
```

Define function. It takes 8 input arguments: 4 strings of length 8 for the first customer and another 4 strings of length 8 for the second customer. This way we can compare 2 customers side-by-side for 32 movies.

The function is stored in a database. You can only do this once; running this code a second time will result in an error: OperationalError: (1304, 'FUNCTION HAMMINGDISTANCE already exists').

To check this function you can run this SQL statement with 8 fixed strings. Notice the "b" before each string, indicating that you're passing bit values. The outcome of this particular test should be 3, which indicates the series of strings differ in only 3 places.

This runs the query.

If all is well, the output of this code should be 3.

Now that we have our hamming distance function in position, we can use it to find similar customers to a given customer, and this is exactly what we want our application to do. Let's move on to the last part: utilizing our setup as a sort of application.

4.5.5 Step 6: Presentation and automation

Now that we have it all set up, our application needs to perform two steps when confronted with a given customer:

- Look for similar customers.
- Suggest movies the customer has yet to see based on what he or she has already viewed and the viewing history of the similar customers.

First things first: select ourselves a lucky customer.

FINDING A SIMILAR CUSTOMER

Time to perform real-time queries. In the following listing, customer 27 is the happy one who'll get his next movies selected for him. But first we need to select customers with a similar viewing history.

Listing 4.13 Finding similar customers

We do two-step sampling. First sampling: index must be exactly the same as the one of the selected customer (is based on 9 movies). Selected people must have seen (or not seen) these 9 movies exactly like our customer did. Second sampling is a ranking based on the 4-bit strings. These take into account all the movies in the database.

Pick customer from database.

```
customer_id = 27
sql = "select * from movie_comparison where cust_id = %s" % customer_id
cust_data = pd.read_sql(sql,mc)
sql =  """ select cust_id,hammingdistance(bit1,
bit2,bit3,bit4,%s,%s,%s,%s) as distance
            from movie_comparison where bucket1 = '%s' or bucket2 ='%s'
or bucket3='%s' order by distance limit 3""" %
      (cust_data.bit1[0],cust_data.bit2[0],
cust_data.bit3[0], cust_data.bit4[0],
        cust_data.bucket1[0], cust_data.bucket2[0],cust_data.bucket3[0])
shortlist = pd.read_sql(sql,mc)
```

We show the 3 customers that most resemble customer 27. Customer 27 ends up first.

Table 4.5 shows customers 2 and 97 to be the most similar to customer 27. Don't forget that the data was generated randomly, so anyone replicating this example might receive different results.

Now we can finally select a movie for customer 27 to watch.

Table 4.5 The most similar customers to customer 27

	cust_id	distance
0	27	0
1	2	8
2	97	9

Finding a new movie

We need to look at movies customer 27 hasn't seen yet, but the nearest customer has, as shown in the following listing. This is also a good check to see if your distance function worked correctly. Although this may not be the closest customer, it's a good match with customer 27. By using the hashed indexes, you've gained enormous speed when querying large databases.

Listing 4.14 Finding an unseen movie

```
cust = pd.read_sql('select * from cust where cust_id in (27,2,97)',mc)
dif = cust.T
dif[dif[0] != dif[1]]
```

Select movies customers 27, 2, 97 have seen.

Transpose for convenience.

Select movies customer 27 didn't see yet.

Table 4.6 shows you can recommend movie 12, 15, or 31 based on customer 2's behavior.

Table 4.6 Movies from customer 2 can be used as suggestions for customer 27.

	0	1	2
Cust_id	2	27	97
Movie3	0	1	1
Movie9	0	1	1
Movie11	0	1	1
Movie12	1	0	0
Movie15	1	0	0
Movie16	0	1	1
Movie25	0	1	1
Movie31	1	0	0

Mission accomplished. Our happy movie addict can now indulge himself with a new movie, tailored to his preferences.

In the next chapter we'll look at even bigger data and see how we can handle that using the Horton Sandbox we downloaded in chapter 1.

4.6 Summary

This chapter discussed the following topics:

- The main *problems* you can run into when working with large data sets are these:
  - Not enough memory
  - Long-running programs
  - Resources that form bottlenecks and cause speed problems
- There are three main types of *solutions* to these problems:
  - Adapt your algorithms.
  - Use different data structures.
  - Rely on tools and libraries.
- Three main techniques can be used to *adapt an algorithm*:
  - Present algorithm data *one observation at a time* instead of loading the full data set at once.
  - *Divide matrices into smaller matrices* and use these to make your calculations.
  - Implement the *MapReduce* algorithm (using Python libraries such as Hadoopy, Octopy, Disco, or Dumbo).
- Three main *data structures* are used in data science. The first is a type of matrix that contains relatively little information, the *sparse matrix*. The second and third are data structures that enable you to retrieve information quickly in a large data set: the *hash function* and *tree structure*.
- Python has many *tools* that can help you deal with large data sets. Several tools will help you with the size of the volume, others will help you parallelize the computations, and still others overcome the relatively slow speed of Python itself. It's also easy to use Python as a tool to control other data science tools because Python is often chosen as a language in which to implement an API.
- The *best practices* from computer science are also valid in a data science context, so applying them can help you overcome the problems you face in a big data context.