UNIT-4

SYLLABUS

Handling large data on a single computer: The problems you face when handling large data

General techniques for handling large volumes of data: Choosing the right algorithm, Choosing the right data structure, Selecting the right tools

General programming tips for dealing with large data sets: Don't reinvent the wheel, Get the most out of your hardware, Reduce your computing needs.

Handling large data on a single computer: -

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.

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.

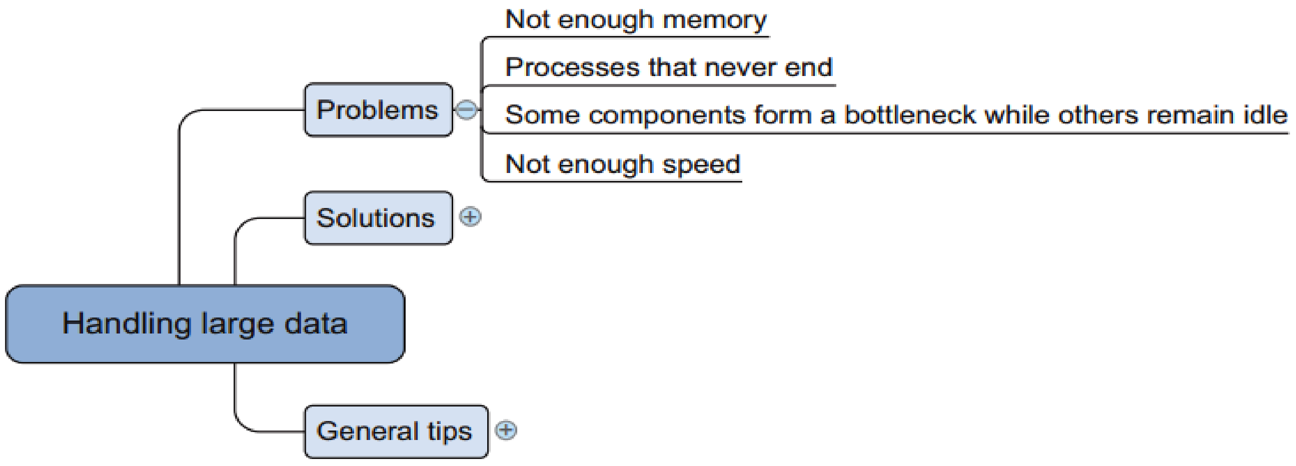


Figure 4.1 Overview of problems encountered when working with more data than can fit in memory

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. 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.

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.

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).

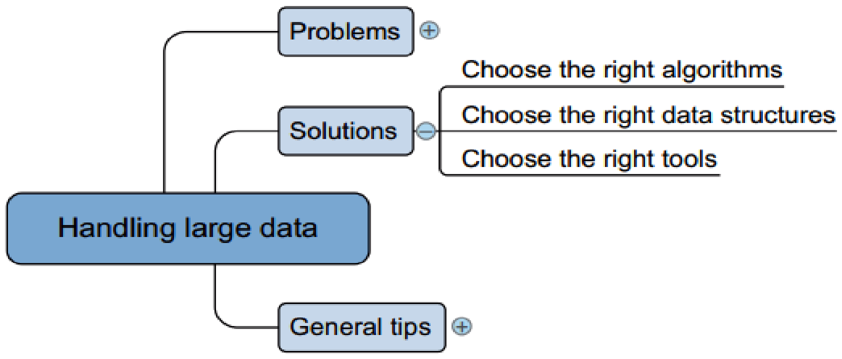


Figure 4.2 Overview of solutions for handling large data sets

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.

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.

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 modernday 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]))
```

The training function.

We start at the first epoch.

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

Adds one to the current number of epochs.

```
    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)
```

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
```

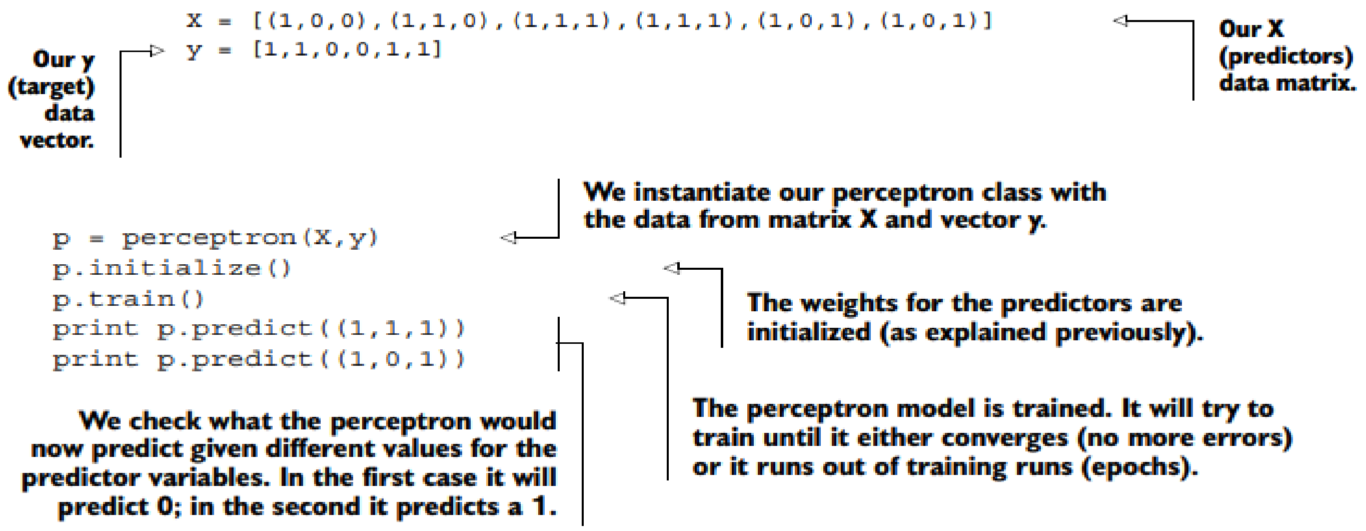
We return the error count because we need to evaluate it at the end of the epoch.

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]
```

Our y (target) data vector.

Our X (predictors) data matrix.

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

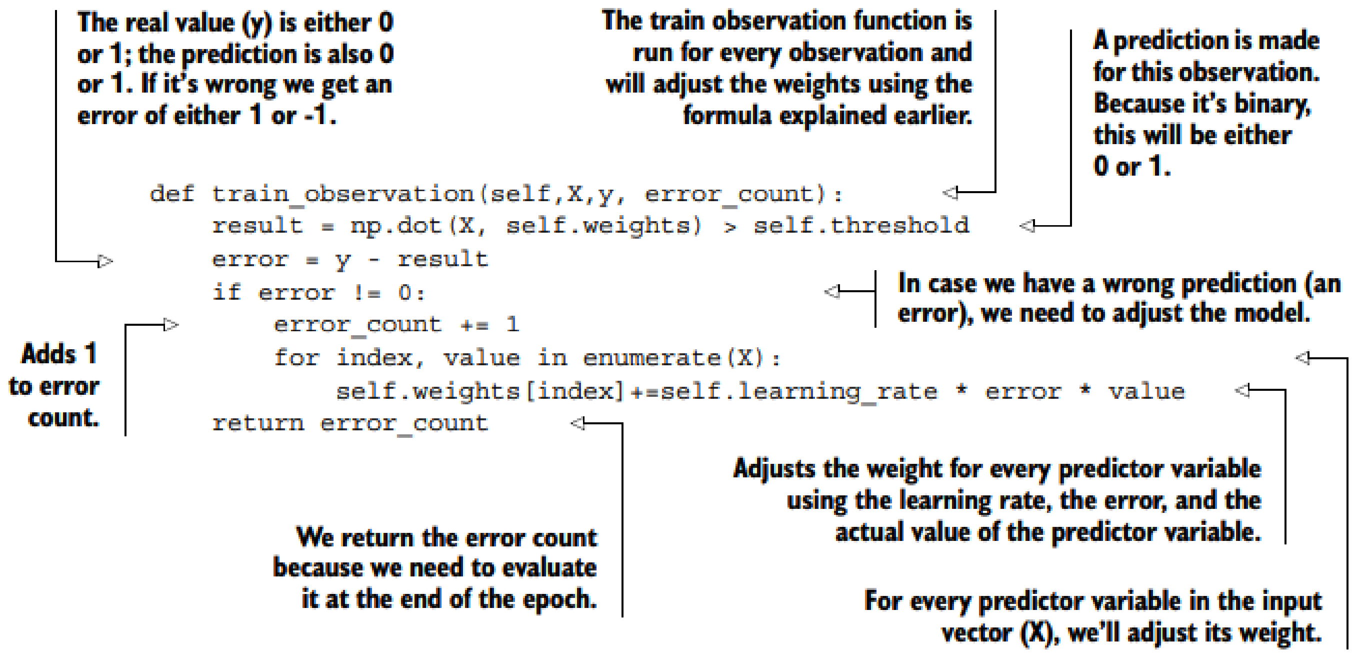
We instantiate our perceptron class with the data from matrix X and vector y.

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 as 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.
- **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: -

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 sub matrices.

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.

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

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

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

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

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')
```

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).

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

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.

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. 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.

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.

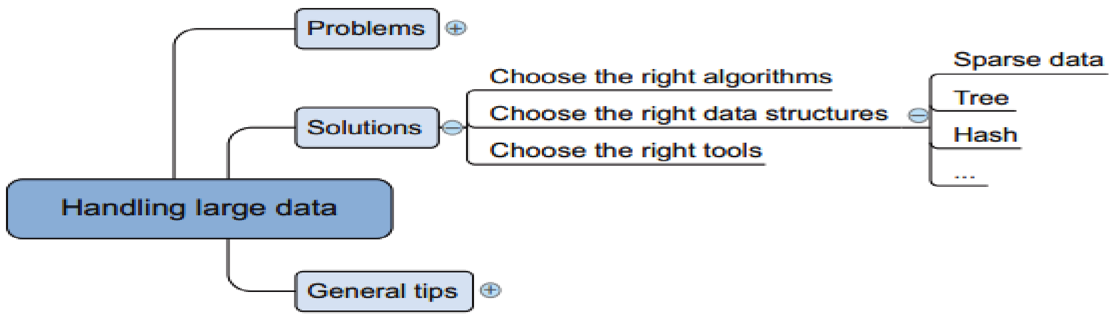
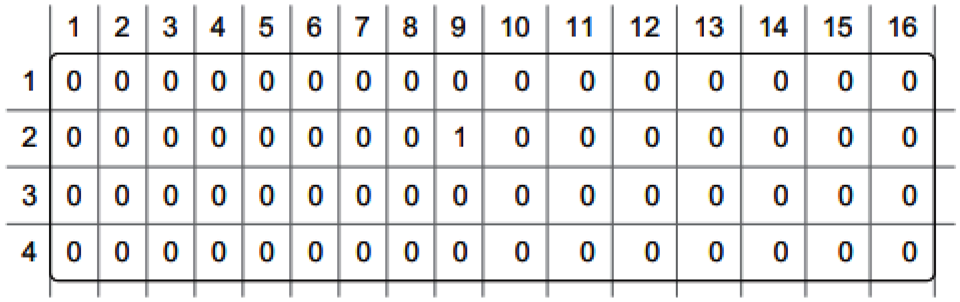


Figure 4.5 Overview of data structures often applied in data science when working with large data

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 tweets. Most of them probably have fewer than 30 words, but together they might have hundreds or thousands of distinct words. In 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.



	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

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 sub trees 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.

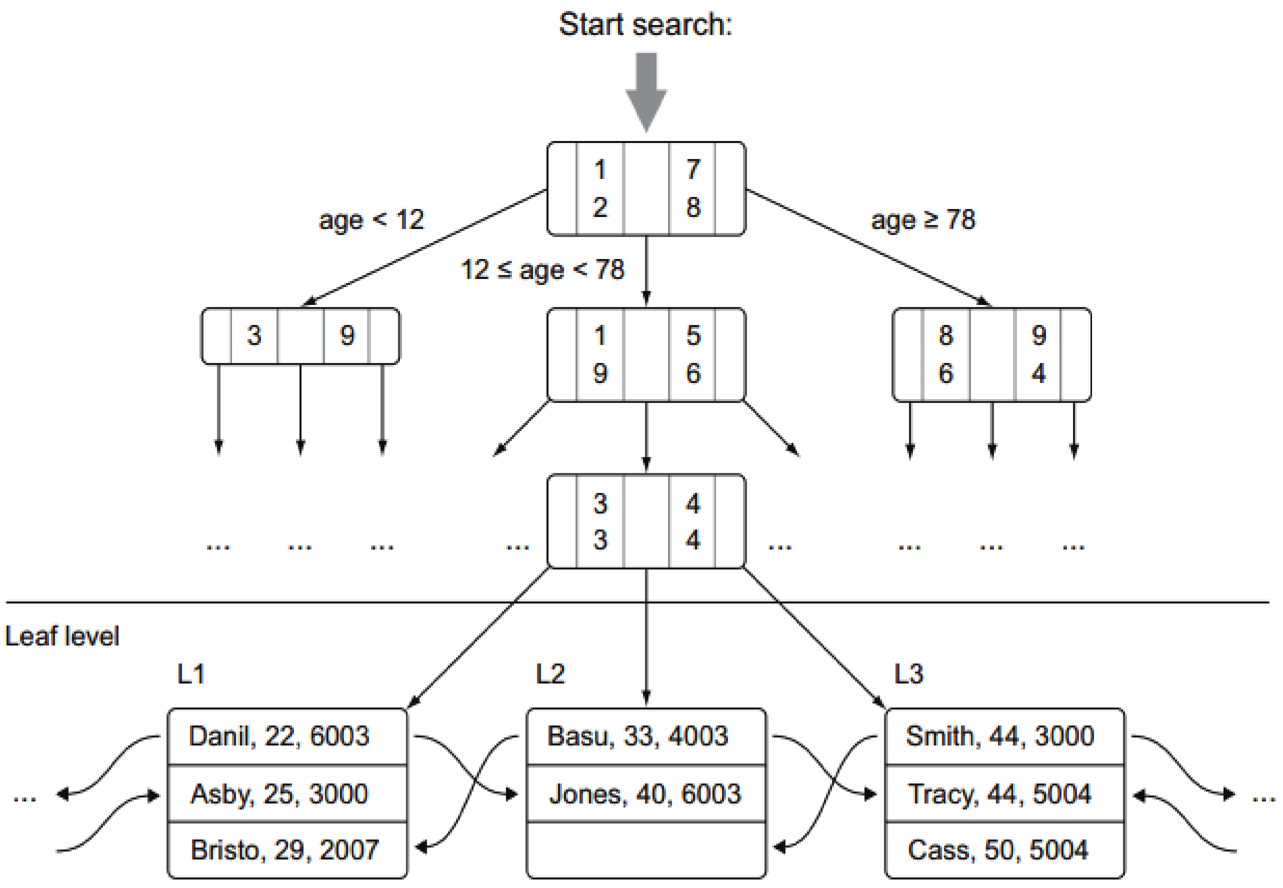


Figure 4.7 Example of a tree data structure: decision rules such as age categories can be used to quickly locate a person in a family tree

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.

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.

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. Hash tables are used extensively in databases as indices for fast information retrieval.

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.

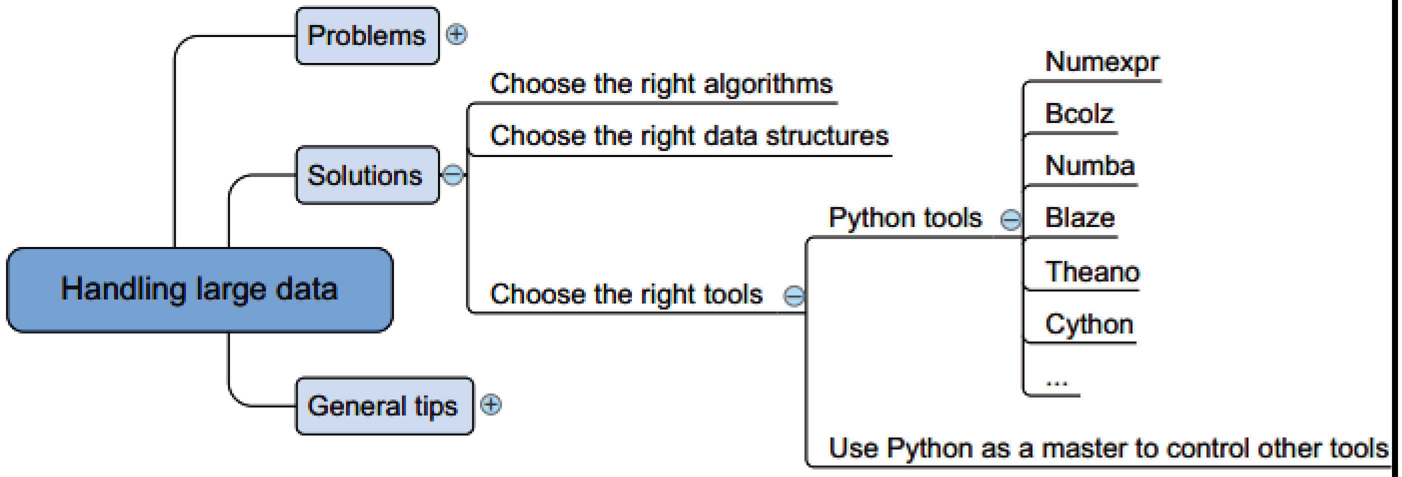


Figure 4.8 Overview of tools that can be used when working with large data

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
- **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.
- 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.
- 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.

- 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.
- 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.
- Dask—Dask enables you to optimize your flow of calculations and execute them efficiently. It also enables you to distribute calculations.

These libraries are mostly about using Python itself for data processing. 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.

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.

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.

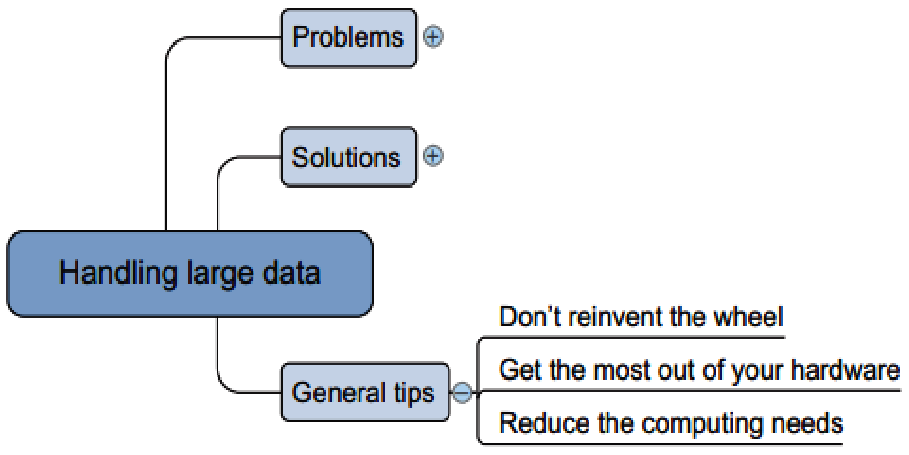


Figure 4.9 Overview of general programming best practices when working with large data

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 machinelearning 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.

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 you're 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.

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.

UNIT WISE IMPORTANT QUESTIONS: -

1. Explain the problems you face when handling large data.
2. Explain the solutions for handling large data sets.
3. Discuss in detail about block matrices
4. What is Map Reduce? Explain in detail.
5. Explain in details about online learning algorithms
6. What are problems encountered when working with more data than can fit in memory? Explain
7. How to choose the right data structure? Explain
8. List and explain different libraries that can help you deal with large data.
9. "Don't reinvent the wheel". Justify the statement
10. Write the possibilities to shift the workload from an overtaxed resource to an underutilized resource.
11. Explain about general programming best practices when working with large data in detail.
12. What are the different techniques to adapt algorithms to large data sets? Explain.