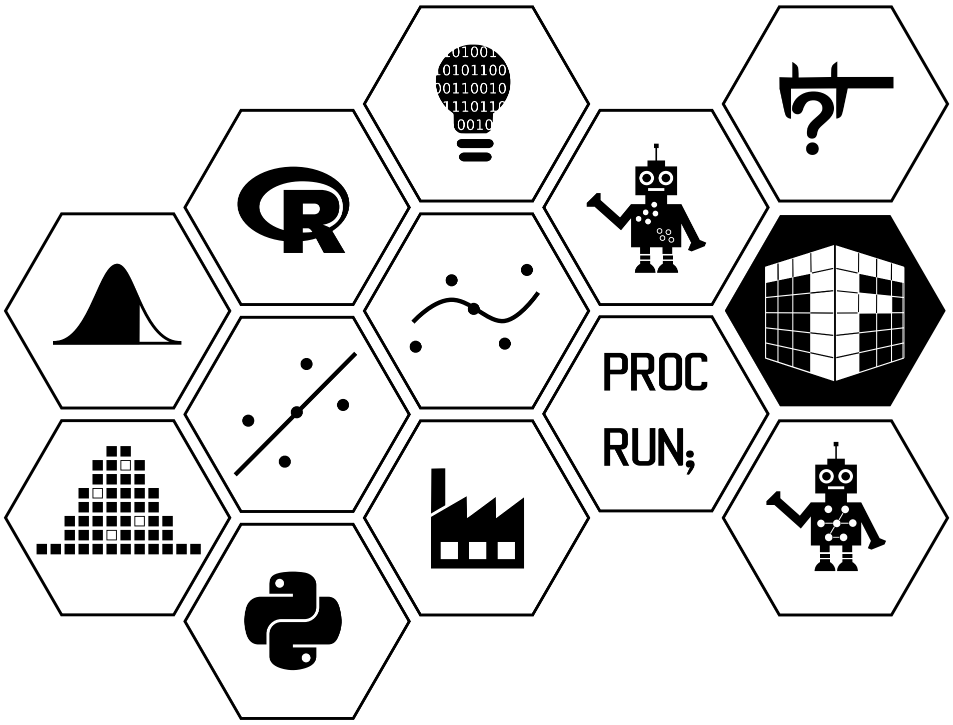
Large-Scale Computing for Data Analytics

Colin Torney and Vinny Davies

Academic Year 2021-22

Week 1:

Scalable Computing and Complexity



General information: Large-Scale Computing for Data Analytics

Course description

This course introduces students to deep learning and convolutional neural networks, and presents an overview of systems for large-scale computing and big data.

Introduction to the Course

https://youtu.be/SbVeO7pmwVI

Duration: 4m11s

Course schedule

The course will consist of ten weeks. The topics are:

- Week 1: Scalable Computing and Complexity
- Week 2: Introduction to TensorFlow
- Week 3: Debugging, Monitoring and Training
- Week 4: Mechanics of TensorFlow
- Week 5: Image classification and the Keras API
- Week 6: Convolutional Neural Networks
- Week 7: Recurrent Neural Networks
- Week 8: TensorFlow Probability
- Week 9: Distributed Large-Scale Analytics
- Week 10: Data Analytics using Spark

Office hours

There are no set office hours during the week, however you can use the moodle forums to ask questions on the forum for each week.

We will also be running live sessions and drop in question-and-answer sessions thoughout the course. Details will be provided on moodle.

Please take advantage of these opportunities to discuss with us any issues you might have about any aspect of the course.

Aims

The aims of this course are:

- to train students in the efficient implementation of computationally expensive data-analytic methods and/or data-analytic methods for big data ;
- to introduce students to deep learning and convolutional neural networks, both in terms of applications and implementation in frameworks such as Tensorflow or Keras; and
- to introduce students to enterprise-level technology relevant to big data analytics such as Spark, Hadoop or NoSQL databases.

Intended Learning Outcomes

By the end of this course you will be able to:

- assess and compare the complexity of an algorithm and implementation both in terms of computational time and memory, as well as suggest strategies for reducing those;
- distinguish between different types of deep and/or convolutional neural networks and choose an appropriate network for a given problem;
- fit a neural network using specialised frameworks such as Tensorflow or Keras and assess the result;
- discuss important methodological aspects underpinning deep learning;

- explain the differences between SQL and NoSQL databases and assess their suitability in different real-life settings; and
- explain the basic concepts underpinning big data systems such as Spark or Hadoop and discuss their suitability and use in different scenarios.

Assessment

This course has three different types of assessment.

- There will be one assignment for this course, worth 20%. The assignment consists of a small number of programming tasks.
- There will also be a project (worth 40%), during which you will work on a bigger, more in-depth task.
- Finally, there will be two quizzes (each worth 20%).

This course has no final degree exam.

Assessment	Weight	Handed Out	Due in
Quiz 1	20%	May 16th	June 5th
Assignment 1	20%	May 30th	June 19th
Project	40%	June 20th	July 25th
Quiz 2	20%	July 11th	August 15th

Books

TensorFlow

There are many books available for learning TensorFlow and the university library has several that are available as ebooks. If you're considering purchasing a book on TensorFlow, be aware that it is evolving fast and many of the code snippets in the books are quickly out-of-date.

Hands-On Machine Learning with Scikit-Learn and TensorFlow

http://shop.oreilly.com/product/0636920052289.do

This book provides an overview of Scikit-Learn and TensorFlow. Although only half the book is about TensorFlow it is a comprehensive introduction and the ebook is available in the University of Glasgow library. The 2nd edition of the book deals with TensorFlow 2 but both editions provide a good overview of key topics.

Pro Deep Learning with TensorFlow: A Mathematical Approach to Advanced Artificial Intelligence in Python

https://www.apress.com/gb/book/9781484230954

This is a good introduction to some of the mathematical concepts that underpin TensorFlow and it is available from the library as an ebook.

Large-scale computing

This course is focusing on large-scale computing and big data. The first eight weeks will focus on large-scale computing and TensorFlow, a library for deep learning. The remaining two weeks will focus on big data. This week will give a short overview of large-scale computing, explain some of the underpinning concepts, and introduce MapReduce, a programming model for working with big data sets.

Background

With the advent of computer-aided decision making and machine learning as well as the increasingly fast process of digitisation the demand for computing power has increased over the last decade and keeps increasing.

Historically, systems with high hardware specifications and parallel computing capabilities, known as *supercomputers*, have been used for meeting the growing demand for computing and storage resources. Supercomputers offer *high performance computing (HPC)* services, but they incur high costs of acquisition and maintenance.

In the meantime, computer hardware has become more powerful and affordable nowadays. As a result, organisations that cannot afford supercomputers can set up less costly *computer clusters* to meet their computing requirements. Even if supercomputers are available, routine computation can be scheduled on computer clusters and only jobs with high performance requirements tend to be sent to supercomputers.

Large-scale distributed computing

However, even though being much less costly than supercomputers, setting up a compute cluster requires a significant investment. Nowadays, users who do not have local access to supercomputers and compute clusters can make use of different types of *large-scale computing* technologies. Grid computing and cloud computing are common examples of such technologies for carrying out computations at large scale.

Grid computing is the result of academic research in the 1990s. A grid brings together resources from different domains under a shared computing environment. An example of a project that made use of such distributed computing is SETI@Home, which started in the late 1990s and allowed users to make their computer available to search radio signals for signs of extraterrestrial intelligence.

Cloud computing

A decade later *cloud computing* started to become increasingly popular. A large number of commercial providers began offering paid-for on-demand access to distributed resources for data storage and computing. Put another way, cloud computing is nothing other than renting compute power and storage from someone else's data centre.

Due to the providers' economies of scale cloud computing can provide a cheaper way of accessing data storage and computing power.

There are several cloud computing platforms, such as

- Amazon Web Services (AWS)
- Google Cloud
- IBM Cloud
- iland by VMWare
- Microsoft Azure
- Qubole, founded by former big data employees of Facebook
- Rackspace
- Service Cloud by Salesforce

Cloud computing is based on the notion of virtualisation. Users of cloud computing run *virtual machines*. A virtual machine (VM) is an emulation of a physical computer.

Features and advantages of cloud computing One of the main characteristics of cloud computing platforms is that they provide the flexibility to scale resources. This is possible because the "computer" a user gets access to is virtual.

Scaling refers to adding or reducing computing power or memory or data storage or network bandwidth to regulate performance. Scaling helps to achieve higher performance in two possibly ways, by scaling up or by scaling out.

- *Scaling up*, known also as *vertical scaling*, means to increase allocation of resources (computing power, memory, data storage or network bandwidth) on an existing VM. For instance, more memory can be added to a VM to make a database server run faster.

- *Scaling out*, also known as *horizontal scaling*, means to increase allocation of resources by adding more VMs to an existing application. For instance, several new VMs with identical configuration can be created to distribute workload across them.

It is possible to reserve resources temporarily for a demanding task and then release them, scaling in or scaling down the application. The flexibility to scale cloud resources according to ongoing needs lends the term *elastic cloud* to cloud computing platforms.

Cloud computing offers a number of advantages and utilities:

- It is a *managed system*, since maintenance and support is made available by the cloud provider.
- It offers *security* due to automatic encryption and other security measures put in place by the provider.
- It enables *scalability* with virtually unlimited computing power and data storage.
- It is *accessible* remotely over HTTP or HTTPS, thus allowing more flexible access virtually from everywhere.
- It is a *durable* infrastructure, which allows *data redundancy* (same data stored in two separate locations) and *data replication* (data copied multiple times across different sites, servers or databases).

Hardware considerations

The electronic circuitry that performs the basic operations a computer program undertakes is called a *processor*. The processor is the engine that carries out mathematical operations, such as arithmetic, it controls logic flows and executes input and output operations.

There are two main types of processing unit available for large scale computing. The hardware configuration can have a large impact on the computational power both for on-site computing clusters and cloud computing.

Central processing unit (CPU). The CPU is the heart of the computer. It performs the majority of operations on a standard PC and is optimised to perform high-precision, fast calculations. A CPU consists of a few cores (up to 32) optimised for sequential serial processing. Many high-performance compute clusters operate by using multiple CPUs in parallel.

Graphical processing unit (GPU). As the name suggest GPUs were originally developed for rendering graphics. They were optimised to perform thousands of lower-precision operations in parallel and so consist of thousands of smaller and more efficient microprocessor cores. While GPUs were originally developed for graphical operations, their architecture make them ideally suited to many parallel programming tasks and they have been used for general purpose computing for many years. GPUs typically have a far higher number of cores than CPUs. Even consumer-grade GPUs can have thousands of cores. Though each core might be, on its own slower, than the core of a CPU, it is their number that makes the difference. Modern GPUs can be used as general-purpose processing units that are 50–100 times faster than CPUs for tasks that require multiple parallel processes. GPUs can be programmed directly using specialised libraries such as CUDA while modern libraries such as TensorFlow take advantage of GPU hardware without requiring the programming to worry about parallel GPU threads or data transfer.

Computational cost and complexity

When working with simple algorithms and models as well as "small" data, it is often not necessary to worry about the cost of an algorithm, either in terms of computing time or in terms of memory requirements. In these cases it is often better to focus on simple and easy-to-maintain implementations, even if these turn out to be less efficient. However, in this course we will be looking at more complex algorithms and models as well as big data. In both cases computational and memory complexity will become important factors.

The notion of complexity is utilised for comparing different algorithms in terms of their run time and storage requirements. Thus, complexity helps form criteria for choosing the most appropriate method to solve a problem under certain circumstances. Algorithms are assessed mainly on the basis of their time and space complexity. *Time complexity* quantifies the amount of time taken by an algorithm to run as a function of its input size. *Memory complexity* (or *space complexity*) quantifies the amount of memory taken by an algorithm to run as a function if its input size.

Often, there is a trade-off between time and space complexity. Equally often, it is not possible to make all operations efficient. Depending on the requirements, one has to trade-off efficiency increases for one type of operation against lower efficiency for other types of operations. For example, when choosing a suitable a data structure one might want to focus on making the insertion of new elements as efficient as possible, but one might alternatively want to focus on the efficiency of looking up values from the data structure. Depending on what our focus is we would use different data structures.

Computational cost

Complexity is seen as a function of input sizes. Often there is just one input size (say the number of data points), and we will refer to it as n. The key questions we are interested in are:

- How much slower will the algorithm get if we increase n, or
- How much more memory will the algorithm need if we increase n?

The complexity is typically expressed in terms of *asymptotic complexity bounds*, which we will look at in more detail later on. The time complexity is typically measured in floating point operations (or FLOPS). Adding two numbers, multiplying them or comparing them are examples of floating point operations.

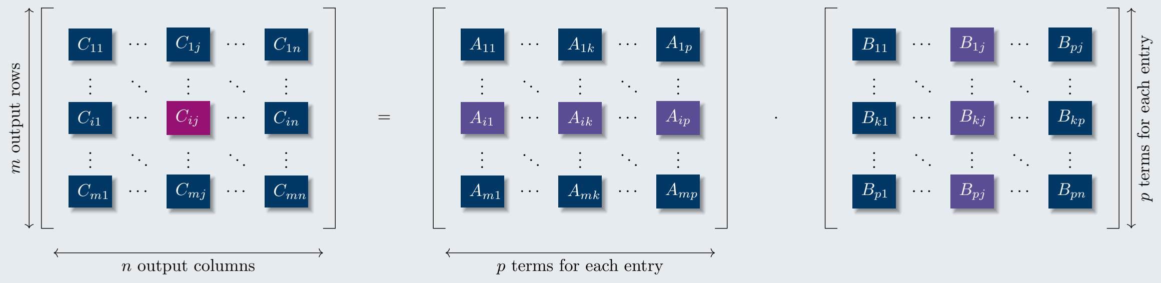
Example 1 (Matrix multiplication).

Suppose we want to multiply two matrices $\mathbf{A}$ and $\mathbf{B}$. Suppose $\mathbf{A}$ is an $m \times p$ and $\mathbf{B}$ is an $p \times n$ matrix.

The resulting matrix $\mathbf{C}$ is of dimension $m \times n$, and each entry is calculated as

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj} = A_{i1}B_{1j} + \ldots + A_{ip}B_{pj}$$

To calculate each entry of $\mathbf{C}$ we need to perform p multiplications and p additions (strictly speaking $p-1$ additions, but coding it up as p additions is easier, see below). So each entry requires $2p$ floating point operations.


m output rows
C_{11} C_{1j} C_{1n}
C_{i1} C_{ij} C_{in}
C_{m1} C_{mj} C_{mn}
A_{11} A_{1k} A_{1p}
A_{i1} A_{ik} A_{ip}
A_{m1} A_{mk} A_{mp}
B_{11} B_{1j} B_{pj}
B_{k1} B_{kj} B_{kp}
B_{p1} B_{pj} B_{pn}
p terms for each entry
n output columns
p terms for each entry

As there are $m \cdot n$ entries in $\mathbf{C}$, the overall cost of calculating $\mathbf{C}$ is $2mnp$.

We can see this if we look at a simple Python implementation.

```python
import numpy as np

m = 10                              # Set up dimensions
n = 8
p = 9
A = np.random.normal(0, 1, (m,p))   # Simulate matrices
B = np.random.normal(0, 1, (p,n))
C = np.zeros((m, n))                # Set up matrix to store result

for i in range(m):                  # Iterate through all entries of C
    for j in range(n):
        for k in range(p):          # Loop to compute the sum
            C[i,j] = C[i,j] + A[i,k] * B[k, j]
```

Of course it would be easier to simply use `C=A @ B`, but "under the bonnet" it would perform effectively the same calculation.

We have calculated the matrix `C` using a triply nested loop. The addition and the multiplication in the inner-most loop get carried out $m \cdot n \cdot p$ times each, giving again the time cost of $2mnp$ floating point operations.

In terms of memory required, all we need is to allocate the $m \cdot n$ matrix $\mathbf{C}$, so the memory complexity is $m \cdot n$.

If both matrices $\mathbf{A}$ and $\mathbf{B}$ are square $n \times n$ matrices (i.e. $m = n = p$), then the computational cost is $2n^3$ floating point operations and the memory cost is n^2.

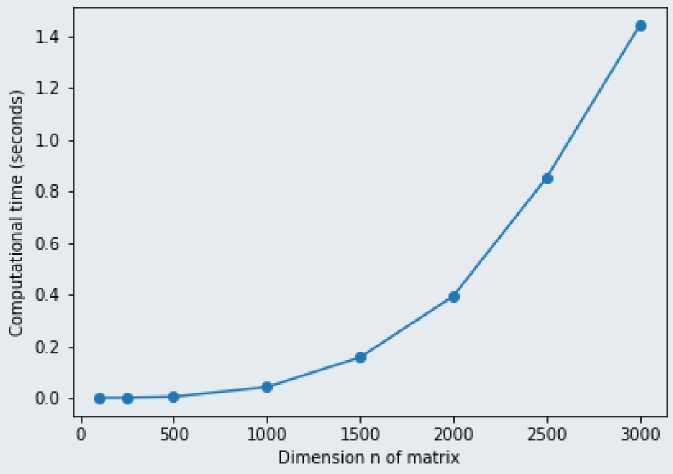
We can "verify" the computational cost by performing a small simulation in Python.

We first define a function that performs the timing for a given value of n …

```python
import numpy as np
import timeit

## Function to time the matrix multiplication
def time_matmult(n, p):
    A = np.random.normal(0, 1, (n, n))
    B = np.random.normal(0, 1, (n, n))
    times = 50                                  # Number of replications
    return timeit.timeit(lambda: A@B, number=times) / times
```

… and then run the timing function for different values on n.

```python
n = np.array([100, 250, 500, 1000, 1500, 2000, 2500, 3000])
times = [ time_matmult(one_n, p) for one_n in n ]
plt.figure()
plt.plot(n, times, '-o')
plt.xlabel("Dimension n of matrix")
plt.ylabel("Computational time (seconds)")
plt.show()
```



We can see that the computation time increases non-linearly with n.

When we look at the computational complexity of an algorithm we are not interested in the exact number of operations, but we want to know how the computational cost increases as we increase (one of) the input dimension(s).

Consider the case of multiplying two $n \times n$ matrices from example 1. We have seen that we require $2n^3$ floating point operations. What happens if we multiply a matrix that is twice as big, i.e. a $(2n) \times (2n)$ matrix? We would require $2 \cdot (2n)^3 = 16n^3 = 8 \cdot (2n^3)$ operations, i.e. 8 times as many as we required for multiplying two $n \times n$ matrices. Because $8 = 2^3$ we would then say that the computational complexity of multiplying two square matrices scales in a cubic manner and say that multiplying two square matrices is an $O(n^3)$ operation.

The basic idea behind the $O(\cdot)$ notation (pronounced "big-oh of …") is that we place a function inside the $O(\cdot)$ that grows as least as fast the computational cost.

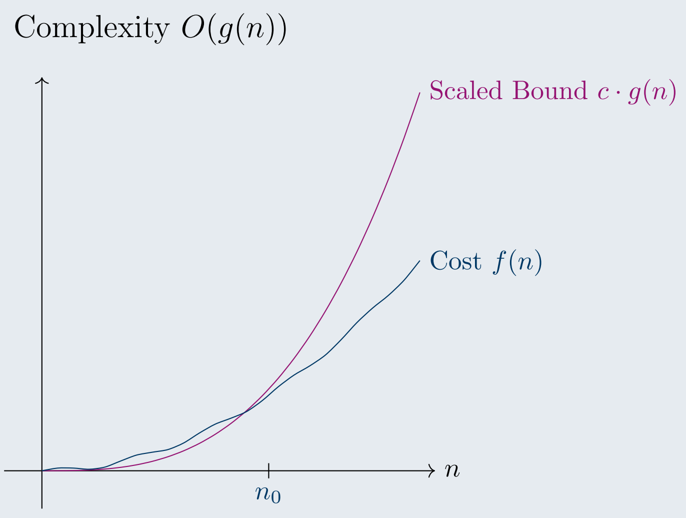
What might be a little surprising is that inverting an $n \times n$ matrix is also $O(n^3)$, i.e. multiplying two matrices has the same time complexity as inverting one of them, even though the latter appears to be much harder for us humans.

Formal definition of the $O(\cdot)$ notation

Consider an an algorithm that requires $f(n)$ operations. We then say that it is of complexity $O(g(n))$ if we can find a positive constant c and a minimum input size n_0 such that for all greater input sizes $n \geq n_0$ we have that $f(n) \leq cg(n)$, i.e. $f(n)$ is dominated by $cg(n)$.

In other words, for an algorithm to be of $O(g(n))$ the bound $g(n)$ must grow faster in n than the number of operations $f(n)$.

In example of multiplying two $n \times n$ matrices (example 1) the number of operations is $f(n) = 2n^3$. This is of $O(n^3)$, as we can choose $n_0 = 1$ and $c = 2$, and obtain $f(n) = 2n^3 \leq 2g(n) = cg(n)$.



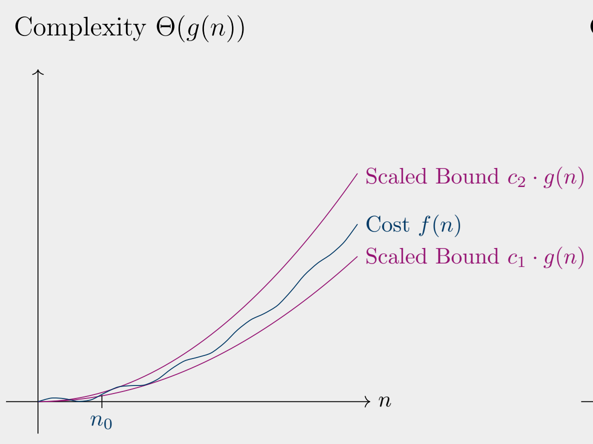
Supplementary material:
$\Omega(\cdot)$ and $\Theta(\cdot)$ notation

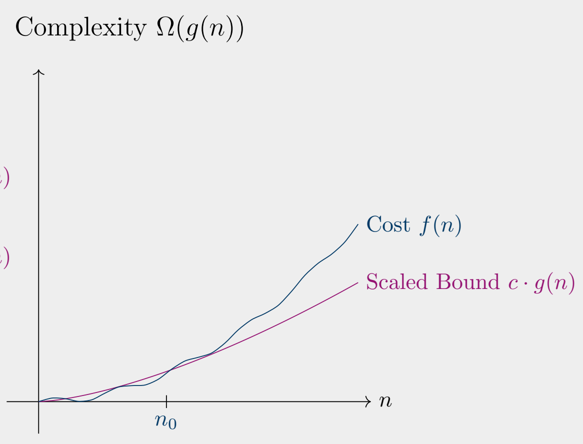
Technically speaking the $O(\cdot)$ notation only provides an upper bound, so multiplying two $n \times n$ matrices is also an $O(n^4)$ operation. Though the $O(\cdot)$ is typically not used this way, we have technically not made a wrong statement.

The $\Theta(\cdot)$ notation avoids this problem. We say that an algorithm is of complexity $\Theta(g(n))$ if we can find two positive constants c_1 and c_2 and a minimum input size n_0 such that for all greater input sizes $n \geq n_0$ we have that $c_1 g(n) \leq f(n) \leq c_2 g(n)$, i.e. $f(n)$ dominates $c_1 g(n)$ and is dominated by $c_2 g(n)$. In other words, for an algorithm to be of $\Theta(g(n))$ the bound $g(n)$ must grow as fast in n as the number of operations $f(n)$.

The reason why the the $O(\cdot)$ notation is more prominent than the $\Theta(\cdot)$ notation is that showing that an algorithm is of complexity $O(\cdot)$ is a lot easier than showing that it is of complexity $\Theta(\cdot)$.

Finally the $\Omega(\cdot)$ notation provides a lower bound on the complexity. We say that an algorithm is of complexity $\Omega(g(n))$ if we can find a positive constant c and a minimum input size n_0 such that for all greater input sizes $n \geq n_0$ we have that $cg(n) < f(n)$, i.e. $f(n)$ dominates $cg(n)$. In other words, for an algorithm to be of $\Omega(g(n))$ the bound $g(n)$ must grow slower in n than the number of operations $f(n)$.





- An $O(1)$ bound means that an algorithm executes in constant time or space irrespective of the size n of the input data.
- An $O(n)$ bound describes an algorithm whose (time or space) performance grows linearly with (in direct proportion to) the size n of the input data.
- An $O(n^2)$ bound refers to an algorithm whose performance grows quadratically with (in direct proportion to the square of) the size n of the input data.
- An $O(n^3)$ bound refers to an algorithm whose performance grows in a cubic manner with (in direct proportion to the cube of) the size n of the input data.

Finally, a problem is said to be NP-hard (non-deterministic polynomial), if we can find no algorithm which has a polynomial run-time everytime, or, to be more precise, there is no algorithm for which there is a polynomial $g(n)$ such that it would be of $O(g(n))$. The travelling salesman problem is an example of an NP-hard problem.

The number of operations required (and thus the run time of an algorithm) is not always deterministic: it often depends on the data. In this case we often look at the average case complexity and the worst-case complexity.

The complexity of a task often also depends on how the data is organised, as the following simple example shows.

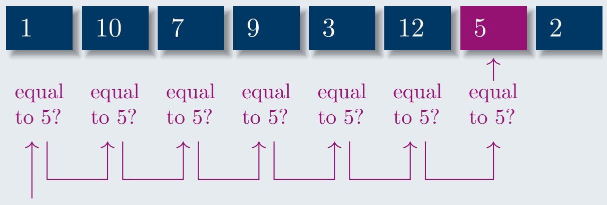
Example 2.

Consider a vector or list of length n containing un-ordered numbers. In Python we could create such a list using

```
l = [1, 10, 7, 9, 3, 12, 5, 2]
n = len(l)
```

Suppose we want to find which entry is 5. All we can do is go through the list one-by-one. What would the time complexity of this be?

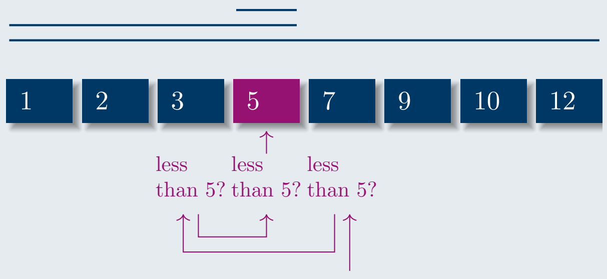
- If we are really lucky, the first entry is the one we are looking for and we are already done. So the best case time complexity is $O(1)$.
- If we are really unlucky, the last entry is the one we are looking for and we need to look at all the entries. So the worst case time complexity is $O(n)$.
- Let's finally look at the average time complexity. With probability $\frac{1}{n}$ the entry we are looking for is in the i-th position, so we need to look on average at $\sum_{i=1}^n ip(i) = \sum_{i=1}^n \frac{i}{n} = \frac{n+1}{2}$ values, so the average case complexity is still $O(n)$.



We can however do better if we know that the entries of the list are sorted.

```
l = [1, 2, 3, 5, 7, 9, 10, 12]
n = len(l)
```

Rather than starting from the beginning, we could pick the entry in the middle (5 or 7, so let's use 7) and compare it to 5. 5 is less than 7, so 5 must come before 7, so we only need to keep what is before 7. Next we compare 5 to 3 (the new entry in the middle), 3 is less than 5, so 5 must come after 3. There is only one entry left, so we found the entry 5 in three steps.



What is the complexity of this algorithm? Every time we make a comparison we can discard half of the remaining vector. If we are unlucky, we need to continue this until we are left with a vector of length 1. So in terms of the number of steps required we need that $2^{\# \text{ steps}}$ exceeds n, i.e. we would need at most $O(\log_2(n))$ steps, which is less than the $O(n)$ from above. So if the data is sorted we can retrieve a value in (average case and worst case) time complexity $O(\log_2(n))$.

In practice, data is rarely sorted. However the same trick can be exploited by what is called indexing (and hashing). If a data structure is indexed, then we additionally maintain a sorted tree of the data which stores where the data is located, so that we can look up values more quickly. In Python, dictionaries and indices of pandas data frames are indexed (hashed). In many relational database systems, columns can be indexed for better performance.

Though indexing increases the speed of retrieving values, it requires additional storage and there is an additional cost when values are added, changed or removed from the list.

Task 1.

Let $\mathbf{A}$ be an $n \times n$ matrix and $\mathbf{w}$ a vector of length n. Explain why the time complexity of multiplying $\mathbf{Aw}$ is $O(n^2)$ and the memory complexity is $O(n)$.

Task 2.

Let $\mathbf{A}$ and $\mathbf{B}$ be matrices of size $n \times n$ and $\mathbf{w}$ a vector of length n. Compare the evaluations $(\mathbf{AB})\mathbf{w}$ and $\mathbf{A}(\mathbf{Bw})$ in terms of time and space complexity.

Task 3.

What is the time and space complexity of the following code? Assume `random()` scales linearly.

```python
import random
a = b = 0

for i in range(n):
    a = a + random.random()
for j in range(m)
    b = b + random.random()
```

Task 4.

Consider a linear regression model of the form

$$Y_i = \beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip} + \varepsilon_i, \qquad i = 1, \ldots, n$$

What is the computational complexity of calculating the least squares estimate

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y},$$

both in terms of the number of observations n and the number of covariates p? Can you verify this empirically in R or Python?

Parallelisation

Normally, the instructions in a programme are executed one-after another (in a *serial* way). An algorithm can run faster if (some of) its steps are executed in parallel. However, this is not always possible. We can only run two or more steps in parallel if they do not depend on each other.

Example 3 (Matrix multiplication).

We can carry out the work required to multiply two matrices $\mathbf{A}$ and $\mathbf{B}$ (of dimensions $m \times p$ and $p \times n$) in parallel. Each entry of the resulting $m \times n$ matrix $\mathbf{C} = \mathbf{AB}$ can be set independently of the other ones, as they do not depend on each other:

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj}$$

This means that all the entries can be set at the same time in parallel. Modern linear algebra implementations exploit this and perform matrix multiplication across more than one core of the CPU.

Example 4 (Cumulative sum).

Suppose we have a vector $\mathbf{x}$ of length n and want to calculate the cumulative sum, i.e. the vector $\mathbf{y}$ with entries

$$y_i = \sum_{j=1}^{i} x_i.$$

We could implement this in a serial manner using

```
import numpy as np

n = 1000
x = np.random.normal(0, 1, n)               # Simulate data
y = np.zeros(n)                             # Create vectro to hold result

cumsum = 0
y[0] = x[0]                                 # Set first entry
for i in range(1, len(x)):                  # Loop through remaining vector
    y[i] = y[i-1] + x[i]                    #    to calculate cumulative sum
```

We cannot parallelise this easily, because we need to know `y[i-1]` when setting `y[i]`. If we were to split the loop into two, we could only start the second half once we have finished the first half, as we need to know what value `y[i-1]` takes, and we will obtain that value only once have dealt with the first half.

Task 5.

How can the calculation of the mean be spread across several parallel processes? Is the same possible for the median?

It is however worth keeping in mind that parallelism incurs an overhead. How large it is depends on the underlying architecture. If parallelising across different cores in a single computer the main overheads stem from coordinating the different threads. However, when parallelising across different nodes in distributed computing environment, sending information from one node to another is typically slow.

In both cases, in order to avoid overheads it is best to parallelise big chunks of code, rather than small chunks and to require a little exchange of information between the parallel threads as possible.

Supplementary material:
Parallelising Bayesian computations

If you have already taken *Uncertainty Assessment and Bayesian Computation* then you have already learned about Markov-Chain Monte Carlo (MCMC) and Sequential Monte Carlo (SMC). MCMC cannot be parallelised as each step depends on the previous step (just like the example of the cumulative sum). All we can do in MCMC is exploit parallelism when calculating the acceptance probability. SMC on the other hand lends itself very well to parallelisation. The particles can be propagated independently of each other in parallel and the nodes only need to communicate when renormalising weight or resampling particles.

Data parallelism

So far, we have only looked at using parallelism to perform a computation more quickly. In data science, a more common occurrence of parallelism is what is called *data parallelism*. In data parallelism, the data is spread across several nodes (computers or servers), typically because the amount of data available is too large for a single node.

The challenge now consists of carrying out the required calculations in a way that

- performs as my operations in parallel as possible, and that
- minimises the amount of data that needs to be transferred between nodes.

Some algorithms (such as linear regression) lend themselves better to such a scenario than others (such as kernelised support vector machines).

Example 5 (Linear regression on two nodes).

Suppose we want to fit a linear regression model of the form

$$Y_i = \beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip} + \varepsilon_i, \qquad i = 1, \ldots, n$$

However the first m observations are stored on the first node and the remaining $n - m$ observations are stored on another node. Let's also assume that the number of covariates is small compared to m or n.

In other words, the first node holds

$$\mathbf{X}_1 = \begin{bmatrix} 1 & x_{11} & \ldots & x_{1p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \ldots & x_{mp} \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$$

The second node holds

$$\mathbf{X}_2 = \begin{bmatrix} 1 & x_{m+1,1} & \ldots & x_{m+1,p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \ldots & x_{np} \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_{m+1} \\ \vdots \\ y_n \end{bmatrix}$$

We have that

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix}$$

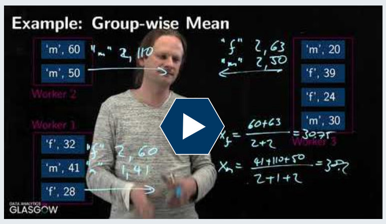
and thus

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} = (\mathbf{X}_1^\top \mathbf{X}_1 + \mathbf{X}_2^\top \mathbf{X}_2)^{-1} (\mathbf{X}_1^\top \mathbf{y}_1 + \mathbf{X}_2^\top \mathbf{y}_2)$$

We can now exploit that we can calculate $\mathbf{X}_1^\top \mathbf{X}_1$ and $\mathbf{X}_1^\top \mathbf{y}_1$ on the first node and, in parallel, $\mathbf{X}_2^\top \mathbf{X}_2$ and $\mathbf{X}_2^\top \mathbf{y}_2$ on the second node. If we assume that the first node is the head node the second node then needs to transfer $\mathbf{X}_2^\top \mathbf{X}_2$ and $\mathbf{X}_2^\top \mathbf{y}_2$ to the first node and the first node can then calculate $\hat{\boldsymbol{\beta}}$ using the formula from above. If p is small only very little data needs to be transferred.

We will now look at data parallelism in practice and a specific programming model for processing data in parallel known as MapReduce.

MapReduce



Introduction to MapReduce

https://youtu.be/hySPcxV1yT8

Duration: 11m55s

If we want to perform computations with the data, we need to perform these computations across a large number of nodes as well. We have already looked at this idea of data parallelism earlier.

MapReduce is a popular programming model for performing calculations in such a scenario and is used in most big data processing frameworks.

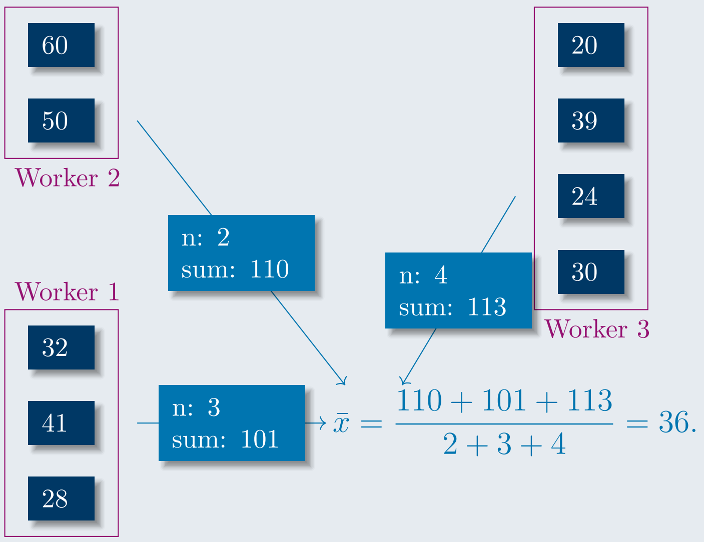
Motivating examples

Before we look at the details of MapReduce, let's look at two simple examples.

Example 6 (Calculating the mean).

Suppose we have data on one variable of interest stored across different nodes. How can we calculate the overall mean?

For each block of the data we can ask one of the nodes storing this block, which we will call a "worker", to return the number of observations on the node and the sum of the variable, which we can then use to calculate the overall mean as illustrated by the figure below.



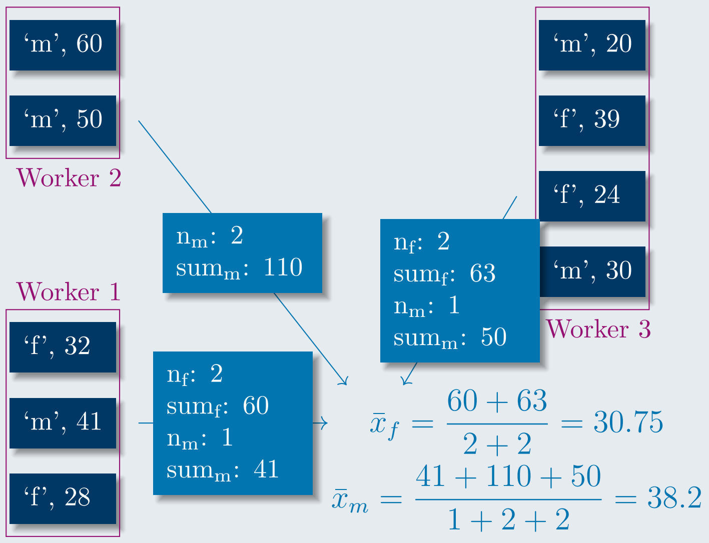
We can calculate the mean very efficiently from the number of observations and the sum of the variable of interest on each worker, as those two numbers contain all the information. Such a data summary that contains all the information required is called a sufficient statistic. We will see later on that methods that have simple sufficient statistics lend themselves very well to efficient data parallelism.

As an aside, we would need to make sure that despite each block being replicated several times (not shown in the above figure) each block will only be processed once. Hadoop will take care of this automatically.

Example 7 (Calculating group-wise means).

Suppose we don't want to just calculate the overall mean, but want to calculate the group means based on an additional categorical variable. For example, imagine that we want to calculate the average age for males and females separately.

In this case, we have to return for each level of the categorical variable, the number of observations having that value and the sum of the corresponding values of the numerical variable.



What we have just seen is a simple example of a MapReduce job.

(1) For an input block of data each worker produced two blocks of information:

- the number of females and the sum of their ages, and
- the number of males and the sum of their ages.

We can view this as key value pairs of the form `[gender, [count, sum_age]]`.

So in other words, for the three blocks in the example we would "emit" following intermediate results:

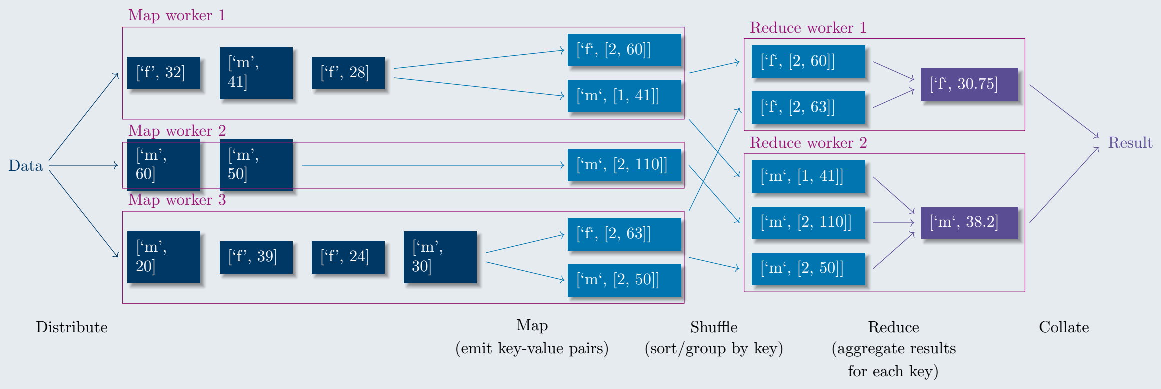
- Worker 1: `[['f', [2, 60]], ['m', [1, 41]]]`
- Worker 2: `[['m', [2, 110]]]`
- Worker 3: `[['f', [2, 63]], ['m', [2, 50]]]`

(2) We can now group the intermediate results by the gender, giving

- Female intermediate data: `[['f', [2, 60]], ['f', [2, 63]]]`
- Male intermediate data: `[['m', [1, 41]], ['m', [2, 110]], ['m', [2, 50]]]`

(3) We can now take each of these and total up the ages and divide the total counts giving for example an average age of $\frac{60+63}{2+2} = 30.75$ for females and $\frac{41+100+50}{1+2+2} = 38.2$.

In the MapReduce paradigm we would refer to (1) as the map step, to (2) as the shuffle step, and to (3) as the reduce step.



The MapReduce paradigm

The MapReduce paradigm consist of applying the steps we have just looked at in example 7.

- The data to be processed is distributed across the cluster to a number of so-called map workers. This might involve moving data between nodes.
- Each map worker reads in the data assigned to it and issues a number of key-value pairs.
- In the shuffle step, the key-value pairs are ordered by the key and all the key-value pairs for one key will be moved to the same node.
- Each of these nodes, called reduce workers, will read in all the key-value pairs for one key, aggregate them and produce an output object.
- These output objects are then collated.

When programming a MapReduce-based algorithm we would need to only implement the map and the reduce step, as these are problem-specific. Hadoop will take care of the remaining steps. Hadoop is written in Java, so native MapReduce jobs in Hadoop need to be coded in Java (or in Pig Latin when using Apache Pig).

Though the MapReduce paradigm is very general and many computations can be rewritten as a MapReduce operation, these might be very inefficient. The smaller the number of key-value pairs emitted in the map step, the less data has to be moved in the shuffle step and the quicker the reduce step will be. So if there low-dimensional summary statistics that contain all the relevant information contained in a block of the data, it is typically best if the map step emits these.

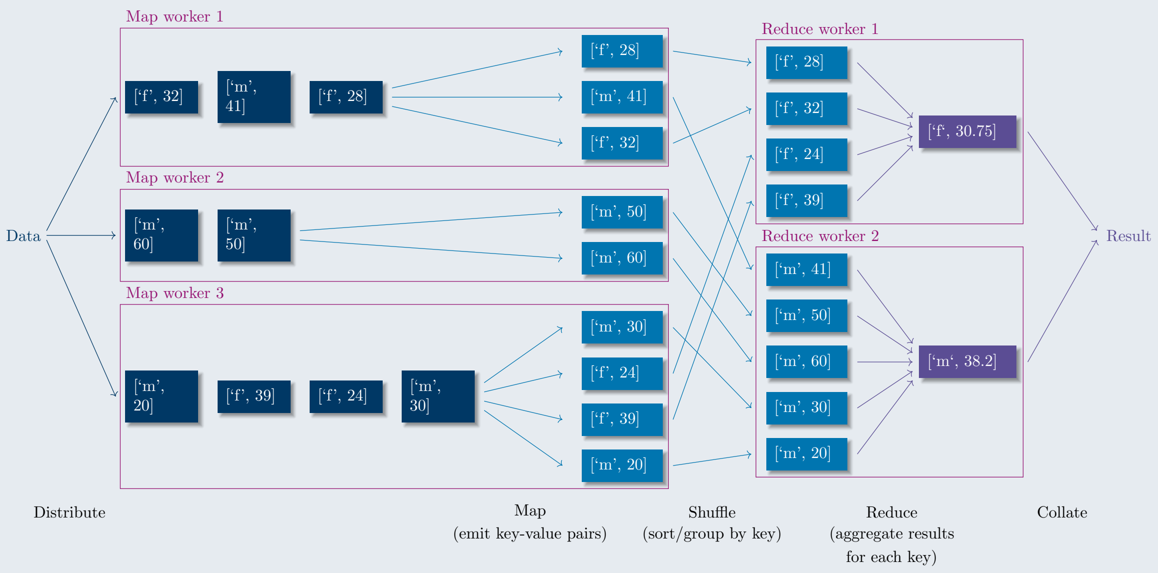
However for a given problem there are often many different ways how that problem can be cast into the MapReduce framework. In practical applications one would need to trade off the effort required to implement different map steps against the computational efficiency of that approach.

Example 8 (Group-wise means revisited).

In the setting of example 7, in which we calculated the average age for both genders, we could also consider the following algorithm, which is simpler to implement, but also less efficient.

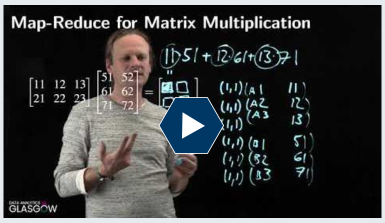
- In the map step we take every observation and emit it as a key-value pair which consists of the gender as the key and the age as the value. So in other words, we simply emit each observation as it is. In contrast to example 7, in which each worker just emitted two key-value pairs, each worker will now emit one key-value pair for each observation.
- The reduce step would simply calculate the average of all observations it is given.

The figure below illustrates this approach.



In example 7 the map step was computationally more complex. In this implementation the map step is much simpler as it only produces labelled observations of the age tagged with the gender. However, this implementation is not efficient at all. We emit many more key-value pairs which will then have to be moved across the network between nodes in the shuffle step, making this implementation very slow.

Though example 8 did not yield a very efficient implementation it showcased a strategy of how many algorithms can be cast into the MapReduce framework. We will employ a similar trick to tackle matrix multiplication.



Matrix multiplication using MapReduce

https://youtu.be/PihmW-5wfxs

Duration: 9m56s

Example 9 (Matrix multiplication).

Consider a $m \times p$ matrix $\mathbf{A}$ and an $p \times n$ matrix $\mathbf{B}$, which we would like to multiply. Suppose that all dimensions are that large that neither $\mathbf{A}$, $\mathbf{B}$ or the result $\mathbf{C} = \mathbf{AB}$ can be stored on a single computer.

The trick involved in turning matrix multiplication into a MapReduce algorithm is to use one reduce worker each the entry of the resulting matrix $\mathbf{C} = \mathbf{AB}$.

This means that the reduce worker will do all the hard work of calculating the numbers in $\mathbf{C}$, and all that map workers have to do is dispatching the input data in a way that it gets sent to the correct reduce worker.

Consider the reduce worker which is meant to calculate the number in row i and column j of the result. It needs to calculate

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{jk} = A_{i1} B_{1j} + \ldots + A_{ip} B_{pj}$$

So this reduce worker needs to obtain the i-th row of $\mathbf{A}$, i.e. A_{i1} to A_{ip} and the j-th column of $\mathbf{B}$, i.e. B_{1j} to B_{pj}. These need to be emitted by the map workers. The key-value pairs need to use (i, j) as the key to make sure the data will be received by the reduce worker computing the (i, j)-th entry of $\mathbf{C}$.

Each key-value pair emitted needs to contain the corresponding entry from the matrix $\mathbf{A}$ or $\mathbf{B}$. We need to include an additional label in the value, so that we know how to use it in the calculation. That label will not be used in the shuffle step, hence we don't include in the key.

For example, we have to make sure that we use A_{i1} in the first term, i.e. multiply it with B_{1j} and not another B_{kj}, so when we emit A_{i1} for example, we also need to include the information that it comes from $\mathbf{A}$ (strictly speaking not required) and that it is meant to enter the calculation of $C_{i,j}$ in the first term of the sum.

To summarise, for the calculation of $C_{(i,j)}$ we need to emit each A_{ik} $(k = 1, \ldots, p)$ as

```
[[i,j], ['A', k, A_ik]
```

where `[i,j]` is the key and `['A',k,A_ik]` is the value.

A_{ik} will not only be used to calculate C_{ij} but also to calculate all the other entries in the i-th row of $\mathbf{C}$, so we need to send C_{ij} to all the n reducers working on the i-th row of $\mathbf{C}$, calculating C_{i1} to C_{in}.

This means that the map workers needs to emit each A_{ik} n times as

```
[[i,j], ['A', k, A_ik]]
```

where j ranges from 1 to n.

Similarly, the map workers need to emit each B_{kj} m times as

```
[[i,j], ['B', k, B_ik]]
```

where k ranges from 1 to m.

The reducers step will receive all the values required to calculate C_{ij}, align them according to the value of k and calculate

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{jk} = A_{i1} B_{1j} + \ldots + A_{ip} B_{pj}.$$

The MapReduce paradigm does not let us label key-value pairs for re-use, so we will emit a lot of redundant data.

In fact, we will emit $2mnp$ key-value pairs, so this MapReduce approach to calculate $\mathbf{C} = \mathbf{AB}$ will require plenty of temporary storage space and might not be fast, but will allow us to multiply two large matrices, each of which might be (much) bigger that what can be stored on a single computer.

We can address the issue of temporary storage space by splitting $\mathbf{C}$ into blocks and using a separate MapReduce jobs for each block, which we can run one after the other.

Example 10 (Linear regression using MapReduce (large n, small or moderate p)).

Consider a linear regression problem with a very large number of observations n.

In linear regression we need to use or calculate the following matrices $\mathbf{X}$ ($n \times p$, or, depending on notation, $n \times (p+1)$) $\mathbf{y}$ ($n \times 1$), $\mathbf{X}^\top \mathbf{X}$ ($p \times p$), $(\mathbf{X}^\top \mathbf{X})^{-1}$ ($p \times p$), $\mathbf{X}^\top \mathbf{y}$ ($p \times 1$) and $\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$ ($p \times 1$).

So for the large n, small or moderate p setting, the computational challenge is in the construction of $\mathbf{X}^\top \mathbf{X}$ and $\mathbf{X}^\top \mathbf{y}$.

We could calculate these using the approach from example 9, but because we know that p is moderate, we can try to perform more calculations inside the map step to obtain a more efficient implementation.

If we denote the i-th row of the covariate matrix $\mathbf{X}$ by $\mathbf{x}_i^\top = (x_{i1}, \ldots, x_{ip})$, then

$$\mathbf{X}^\top \mathbf{X} = \sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^\top$$

In other words the $(j_1, j_2) - th$ entry of $\mathbf{X}^\top \mathbf{X}$ is

$$(\mathbf{X}^\top \mathbf{X})_{j_1, j_2} = \sum_{i=1}^{n} x_{ij_1} x_{ij_2}$$

If the data is now processed by different mappers, each mapper gets a block of the data to processed. We can rewrite

$$(\mathbf{X}^\top \mathbf{X})_{j_1, j_2} = \sum_{\text{blocks } i} \sum_{\text{ in block}_l} x_{ij_1} x_{ij_2}$$

This suggests that the l-th mapper can calculate $\sum_{i \text{ in block}_l} x_{ij_1} x_{ij_2}$ for $j_1, j_2 \in \{1, \ldots, p\}$ and emit the entries using (j_1, j_2) as key.

Similarly, we can calculate $\mathbf{X}^\top \mathbf{y}$ which has j-th entry

$$(\mathbf{X}^\top \mathbf{y})_j = \sum_{i=1}^{n} \mathbf{x}_{ij} y_i = \sum_{\text{blocks } i} \sum_{\text{ in block}_l} x_{ij} y_i$$

This suggests that the l-th mapper for this part of the calculation can calculate $i \text{ in block}_l x_{ij} y_i$ for $j \in \{1, \ldots, p\}$ and emit the entries using j as key.

For both calculations the reducers would simply need to add up all the terms they are assigned as their role is just to implement the out-most sum of the above formulae for $(\mathbf{X}^\top \mathbf{X})_{j_1, j_2}$ and $(\mathbf{X}^\top \mathbf{y})_j$.

If p is moderate the remainder of the calculations, notably

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

can be carried out on a single node. If p is however large, then this would not be possible, and which case the calculation of $\hat{\boldsymbol{\beta}}$ would need to be spread across the cluster. Rather than using direct linear algebra to calculate $\hat{\boldsymbol{\beta}}$ it would in the large p case be more efficient to use an indirect method such as (stochastic) gradient descent.

Looked at from a statistical point of view, when implementing an inferential algorithm it is always best if the map step produces a low-dimensional sufficient statistic that contains all the relevant information from that block of observations. This is what we have done in example 10 for linear regression.

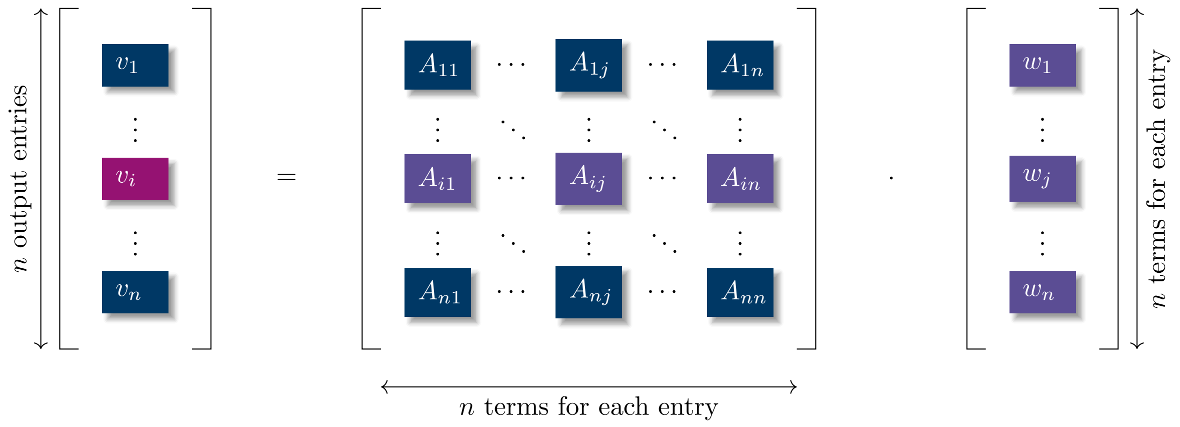
Task 6.

Consider a large text document. Construct a MapReduce algorithm that counts how often each word occurs in the document.

Answers to tasks

Answer to Task 1. $\mathbf{Aw}$ is a vector of length n, so we need to calculate n elements. The i-th element pf $\mathbf{Aw}$ is given by the sum

$$\sum_{j=1}^{n} A_{ij}w_j = A_{i1}w_1 + \ldots + A_{in}w_n.$$



Each entry requires $2n$ floating point calculations, so calculating the entire vector requires $2n^2$ floating point operations. Thus the time complexity of the matrix-vector product $\mathbf{Aw}$ is $O(n^2)$.

The matrix-vector product $\mathbf{Aw}$ yields a vector of length n, therefore it has space complexity $O(n)$.

Answer to Task 2. $(\mathbf{AB})\mathbf{w}$ requires us to compute the matrix product $\mathbf{U} = \mathbf{AB}$, which takes $O(n^3)$ time, and subsequently the matrix-vector product $\mathbf{Uw}$, which takes $O(n^2)$ time. So, the time cost of $(\mathbf{AB})\mathbf{w}$ is $O(n^3 + n^2)$ or, by excluding trailing terms (which grow slower than n^3), $O(n^3)$.

In terms of memory, we need to allocate the temporary $n \times n$ matrix $\mathbf{U}$ and the result vector $\mathbf{w}$ of length n. Thus the memory complexity is $O(n^2 + n)$ or, by excluding trailing terms, $O(n^2)$.

On the other hand $\mathbf{A}(\mathbf{Bw})$ requires to compute the matrix-vector product $\mathbf{v} = \mathbf{Bw}$, which runs in $O(n^2)$ time, and the matrix-vector product $\mathbf{Av}$, which also runs in $O(n^2)$ time. So, the time cost of $\mathbf{A}(\mathbf{Bw})$ is $O(2n^2)$ or, by ignoring scaling factors, $O(n^2)$. Thus, it is less time consuming to compute $\mathbf{A}(\mathbf{Bw})$ instead of $(\mathbf{AB})\mathbf{w}$.

In terms of memory, we need to allocate the temporary vector $\mathbf{v}$ of length n and the result vector $\mathbf{w}$ of length n. Thus the memory complexity is $O(2n)$ or, by ignoring scaling factors, $O(n)$.

So, both in terms of time and memory complexity the second approach is to be preferred over the first one.

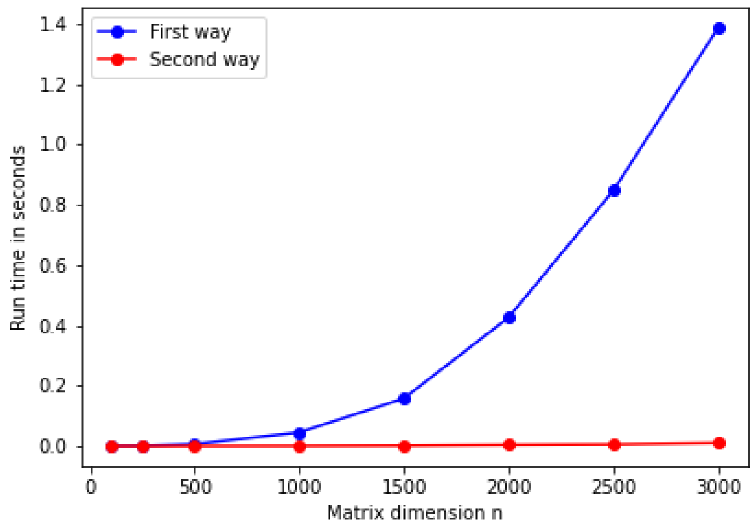
$$= \begin{bmatrix} U_{11} & \cdots & U_{1n} \\ \vdots & \ddots & \vdots \\ U_{n1} & \cdots & U_{nn} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

$$\begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & \cdots & B_{1n} \\ \vdots & \ddots & \vdots \\ B_{n1} & \cdots & B_{nn} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

$$= \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

To demonstrate the difference in time complexity between $\mathbf{A}(\mathbf{B}\mathbf{w})$ and $(\mathbf{A}\mathbf{B})\mathbf{w}$, we can compare the run times in Python for different values of n.

```
import numpy as np
import timeit
import matplotlib.pyplot as plt

n = np.array([100, 250, 500, 1000, 1500, 2000, 2500, 3000])
                                     # Set input dimensions
runtime1 = np.zeros(n.shape)         # Create vectors to hold
runtime2 = np.zeros(n.shape)         #   timings

for i in range(len(n)):              # For each n ...
    A = np.random.normal(0, 1, (n[i], n[i]))
    B = np.random.normal(0, 1, (n[i], n[i]))
    w = np.random.normal(0, 1, n[i]) # ... generate data
    def first_way():                 # Define functions
        return (A@B)@w               #   for timings
    def second_way():
        return A@(B@w)
    times = 50                        # Number of replications
    runtime1[i] = timeit.timeit(first_way, number=times) / times
    runtime2[i] = timeit.timeit(second_way, number=times) / times
                                      # Perform timing

plt.figure()
plt.plot(n, runtime1, 'b-o', label='First way')
plt.plot(n, runtime2, 'r-o', label='Second way')
plt.xlabel("Matrix dimension n")
plt.ylabel("Run time in seconds")
plt.legend()
plt.show()
```



Answer to Task 3. The snippet takes $O(n+m)$ time, as dictated by the two `for` loops. The space complexity of the snippet is constant ($O(1)$), since `a` and `b` are allocated once at the beginning and they do not get reallocated inside the `for` loops.

Answer to Task 4. $\mathbf{X}$ is a $n \times (p+1)$ matrix and $\mathbf{y}$ is a vector of length n.

Calculation	Computational cost
Multiplying $\mathbf{X}^\top$ and $\mathbf{X}$	$O(np^2)$
Multiplying $\mathbf{X}^\top$ and $\mathbf{y}$	$O(np)$
Inverting $\mathbf{X}^\top\mathbf{X}$	$O(p^3)$
Multiplying $(\mathbf{X}^\top\mathbf{X})^{-1}$ and $\mathbf{X}^\top\mathbf{y}$	$O(p^2)$

Instead of the last two steps it is a little faster to solve the system of equations $\mathbf{X}^\top\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^\top\mathbf{y}$, though this is also $O(p^3)$ (but with smaller constants).

Thus overall computational complexity is $O(np^2+p^3)$. Because we need $n \geq p+1$ in order to be able to calculate unique estimates, np^2 dominates p^3, thus the overall computational complexity becomes $O(np^2)$, as n needs to increase with p.

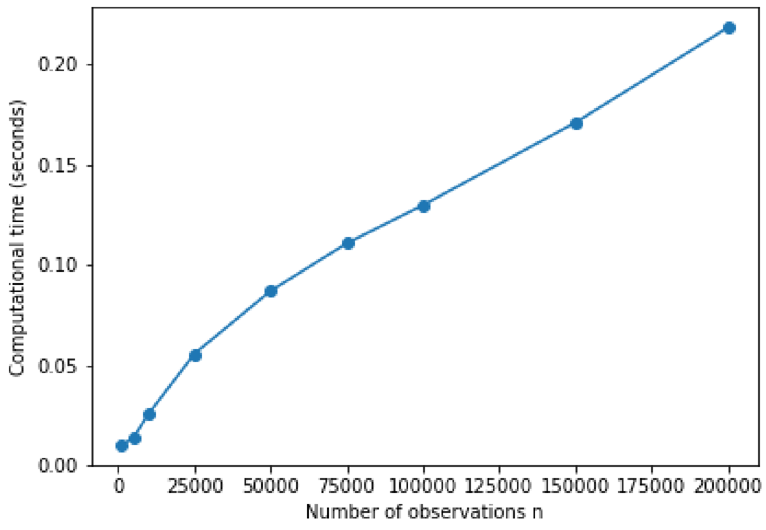
- If we only look at the effect of the number of observations n (keeping the number of covariates p fixed), then the computational complexity is $O(n)$, i.e. linear regression scales linearly in the number of observations.
- If we only look at the effect of the number of covariates p (keeping the number of observations n fixed), then the computational complexity is $O(p^3)$, i.e. linear regression scales in the number of covariates in a cubic way.

We can verify this empirically. We start with the imports and define three functions.

```
import numpy as np
import matplotlib.pyplot as plt
import timeit

# Function to simulate data for linear regression
def simulate_data(n, p):
    X = np.random.normal(0, 1, (n, p))           # Simulate X
    X = np.concatenate((np.ones((n, 1)), X), 1)  # Add column of 1s
    y = np.random.normal(0, 1, (n))              # Simulate y
    return X, y

# Function to calculate least-squares estimate
def compute_beta_hat(X, y):
    XtX = X.transpose() @ X
    Xty = X.transpose() @ y
    return np.linalg.solve(XtX, Xty)

# Function to time the calculation of the least-squares estimate
def time_regression(n, p):
    X, y = simulate_data(n, p)                      # Simulate data
    def perform_computation():                      # Define function used
        compute_beta_hat(X, y)                      #   in the test
    times = 50                                      # Number of replications
    return timeit.timeit(perform_computation, number=times) / times
```

We first look at the effect of the number of observations n.

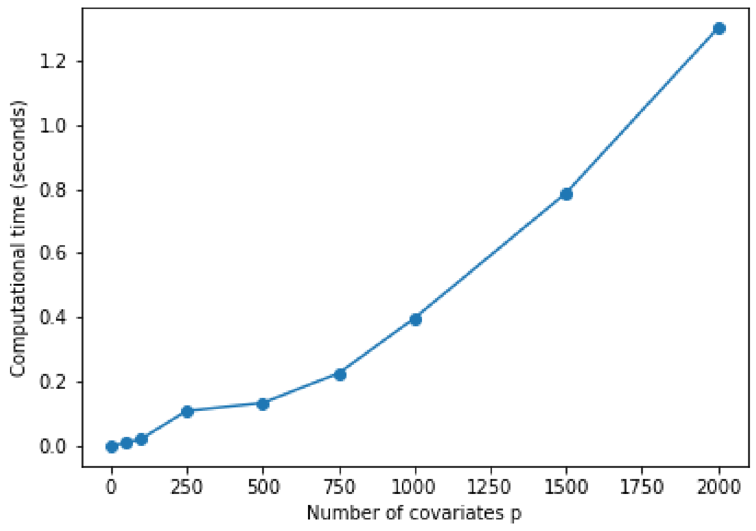
```
n = [1000, 5000, 10000, 25000, 50000, 75000, 100000, 150000, 200000]
p = 100
times = [ time_regression(one_n, p) for one_n in n ]
plt.figure()
plt.plot(n, times, '-o')
plt.xlabel("Number of observations n")
plt.ylabel("Computational time (seconds)")
plt.show()
```



We can see the CPU time increases linearly as we increase the number of observations n.

We look at the effect of the number of covariates p next.

```
n = 10000
p = [1, 50, 100, 250, 500, 750, 1000, 1500, 2000]
times = [ time_regression(n, one_p) for one_p in p ]
plt.figure()
plt.plot(p, times, '-o')
plt.xlabel("Number of covariates p")
plt.ylabel("Computational time (seconds)")
plt.show()
```



We can see the CPU time increases in a non-linear (cubic) way as we increase the number of covariates p.

Answer to Task 5. In order to calculate the mean we need to sum up all the values in the vector. This sum can be split in parts and each part carried out independently, so we can easily parallelise the calculation of the mean (though unless we really have a lot of data (that is possibly event distributed between several nodes), there would be little point in doing so, as calculating the mean is quick).

The calculation of the median can however not be easily parallelised. Unless the data is distributed there is little point in trying to calculate the median in parallel.

Answer to Task 6. For each block of text, the map-worker counts for each word in the block how often it occurs and emits key-value pairs in the form `[word, n_occurences]`. The reducer will then simply total up the number of occurrences from the different blocks.

Alternatively, we would simply issue each word in the document as a key-value pair of the form `[word, 1]`. This would require less effort in the implementation, but would yield a less efficient algorithm.