A Taste of Haskell

An Introduction to Functional Programming (and Big Data)

Essential Reading (Homework)

What is a function? Read this Maths Is Fun page on functions: https://www.mathsisfun.com/sets/function.html

Then do the questions at the bottom of the page.

Your answers to the questions:

_____

_____

Read the Maths Is Fun page on domain, range and codomain: https://www.mathsisfun.com/sets/domain-range-codomain.html

Again, do the questions at the bottom of the page.

Your answers to the questions:

_____

_____

What is Functional Programming? (Slideshow) Characteristics of Functional Programming

- Immutability
- Statelessness
- First class functions

Getting Started with Haskell

Objectives

- Gain familiarity with the WinGHCi user interface and learn how to evaluate expressions and edit and compile scripts
- Learn and use some of the fundamental operators and functions in the Haskell Prelude library
- Understand and practice using lists and associated functions
- Learn the syntax for defining simple functions in Haskell
- Homework: read about data types and classes

Your teacher will give you a brief demonstration of the WinGHCi interface: it's very simple.

Simple Expressions

Try entering the following simple expressions:

Expr	Output	Notes
6 + 7	13	We can enter numbers into haskell and it'll run calculations
3 * -6	-18	
3 == 3	True	Equality comparisons can [sometimes] be done with ==
3 /= 3	False	Not equal to is done with /=
not False	True	Boolean not operator
True && True	True	Boolean and operator
True || False	True	Boolean or operator

Standard Functions

Now try calling some functions. These are all defined in the standard Haskell library which is called Prelude:

Expr	Output	Notes
succ 3	4	succ means "successor"
pred 3	2	pred means "preceding"
min 12 56	12	min means "minimum"
max 4 13	13	who would'a guessed?
succ 4 * 3	15	* is left associative
succ (4 * 3)	13	we can use brackets to change that
succ $ 4 * 3	13	the $ operator has lowest precedence
div 5 2	2	division is integer division
mod 5 2	1	mod is modulo

Lists

Lists are very important data structures in Haskell. They are similar to arrays in Pascal and other languages.

A list consists of an ordered collection of values, separated by commas and enclosed in square brackets:

```
myList=[3, 5, 2, 8, 6]
```

A list is **dynamic**: it doesn't have a fixed size but can grow and shrink as required. All the values in a list must be of the same type but this type does not need to be declared explicitly.

Haskell provides lots of useful library functions to manipulate and process lists.

Try these for yourself:

Expr	Output	Notes
[1,2,3] ++ [4,5,6]	[1,2,3,4,5,6]	++ is list concatenation
6:[7,8,9]	[6,7,8,9]	a:[b] is equivalent to [a] ++ [b]
'a':"bcd"	"abcd"	although my version of hs has a String type, it's just a [Char]
"sheep" !! 3	'e'	!! gets the item in position n from a list (like nix' elemAt)
[1..10]	[1,2,3,4,5,6,7,8,9,10]	[a..b] defines a range
head [1..3]	1	head gets the first item in a list
tail [1..3]	[2, 3]	tail gets all but the first item in a list
last [1..3]	3	last gets the last item in a list
length [1..3]	3	length gets the length of a list
reverse [1,2,3]	[3,2,1]	reverse reverses a list
sum [1..10]	55	sum sums through a list
product [1..10]	3628800	if sum is $\sum$, this is $\prod$
take 2 [1,2,3]	[1,2]	gets the first n items in a list
drop 2 [1,2,3]	[3]	gets everything past the first n items in a list
maximum [1..3]	3	like max but for lists
minimum [1..3]	1	like min but for lists

List Comprehensions

Remember Set Comprehensions from last year?

> $\{x |\ x \in \mathbb{N} \ \wedge \ x < 5\}$ means the set of all natural numbers less than 5.

You can build lists in Haskell in a very similar way:

> [x|x<-[1..100],x<5]

-

builds a list drawn from the list [1..100] whose elements are less than 5.

Task

Write List Comprehensions that generate the following lists:

[2,4,6,8,10]

```
[x|x<-[2..10],even x]
```

[1,3,5,7,9]

```
[x|x<-[1..9],odd x]
```

Writing your first functions

Use a text editor to create a file called *first.hs*, which contains the following single line of code:

double x = x + x

Open the file in WinGHCi.

Call your function:

double 21

Easy!

Now add another function to the file (start on a new line):

doubleSum x y = x * 2 + y * 2

Reload the file in WinGHCi and test the new function:

doubleSum 4 5

A file can contain any number of functions and they can be in any order.

Task

Modify your *doubleSum* function so that it calls *double* to do the doubling.

Solution

```
doubleSum x y = double x + double y
```

Task: make a factorial function. Hint: this is very easy if you use one of the standard functions introduced earlier.

```
fact:: Int -> Int
fact x = if x == 1 then 1 else x * fact (pred x)
```

or

```
fact2 :: Int -> Int
fact2 x = product [1..x]
```

Another example, this time a function that takes a list as its input and returns the mean

```
mean xs = sum xs `div` length xs
```

Points to note:

- It's traditional to use "xs", that is a plural, to indicate that an input is a list.
- The *div* function can either be used:
  - as an infix, as above, in which case *div* must be enclosed in **back quotes**
  - as prefix like this: div (sum xs) (length xs)

Task: Make your own *last* functions

Define a function called *mylast* which uses some of the standard list processing functions to duplicate the behaviour of the standard *last* function.

```
last1 xs = xs !! (length xs - 1)
```

Now define another, different function that also mimics *last*.

```
last2 xs = head $ drop (length xs - 1) xs
```

Homework

- Read the "Essential Reading" section in the next chapter: "Data Types and Classes"
- Do the Self-Test exercises on below.

Self-Test: bring your answers to the next lesson You can use the Haskell compiler to check your answers.

1. What are the types of the following values:

[True, True, False]

[Bool]

“cat”

String or [Char]

[‘c’, ‘a’, ‘t’]

String or [Char]

[“cat”,“dog”,“sheep”]

[String] or [[Char]]

[tail, reverse]

[[a] -> [a]]

---

2. Write function definitions, including type declarations, for the following:

   a. A function that returns the second element of a list

```
second :: [a] -> a
second list = list !! 1
```

   b. A function that determines whether or not a list is a palindrome

```
second :: [a] -> a
second list = list !! 1
```

Hint: these can both be done in a single line by using standard list functions from page 5.

Prime Number Generator

Objectives

- Analyse a problem and identify how it can be broken down into a set of functions
- Develop and implement the required functions
- Homework: read about and practice using selection structures in Haskell

Write a Haskell program that will display all the prime numbers up to a specified value.

```
isPrime :: Int -> Bool
isPrime n = length [x|x<-[2..n-1],mod n x==0] == 0
```

```
nextPrime :: Int -> Int
nextPrime n = if isPrime $ succ n then succ n else nextPrime $ succ n
```

Essential Reading – Data Types and Classes

Haskell, like Pascal and Java is a strongly, statically-typed language. Every expression has a type assigned to it at compile time which cannot be changed. If you attempt to break the rules, for example by dividing a string by a number, then the compiler will reject your code. Unlike Pascal and Java, (but like Python) Haskell **infers** types, that is to say it automatically works out the types of expressions without the programmer having to explicitly declare them.

Common types include: *Bool, Char, Int* and *Float*. (Note that type names must start with an Upper case letter)

You can see Haskell's type inference in action by using the :t command which tells you the type of any expression:

>:t True

True :: Bool

>:t "sheep"

"sheep" :: [Char]

Types are also grouped together in **classes**. For example the class *Num* includes the numeric types such as *Int* and *Float*.

>:t 6

6 :: Num a => a

In the example above, Haskell has inferred that 6 is a numeric type but it has not bound it strictly to a specific one: it could be an integer or a float.

Both the inputs and outputs of functions have types. For example the *not* function both takes in and returns a Boolean:

>:t not

not :: Bool -> Bool

Many functions are quite flexible and can operate on whole classes rather than single types. For example the *head* function can take as its input a list of any type and return its first element:

>:t head

head :: [a] -> a

In the example above *a* is called a **type variable**. It means that *head* can operate on a list of any type. (What are the advantages of this?)

When defining your own functions it is good practice to explicitly declare the type of the function's inputs and outputs as the first line of the definition. This makes your code more readable and less likely to generate errors.

For example the function *double* you wrote last lesson, takes in any numeric type and returns a value of that same type. This can be written as:

double:: Num a => a -> a

You can read this as "this function is called double and it takes as its input any numeric type ("a") and returns a result of the same type". You don't have to use *a* as the name of the type variable but it's common practice.

Then you can follow that by the implementation statements. In this case the whole function definition becomes:

double :: Num a => a -> a

double x = x + x

Task – Prime Number Generator

Write a Haskell program that will display all the prime numbers up to a specified value.

Breaking the problem down into functions In your triangles, discuss how you could solve this problem and what functions you will need to write.

Now write code to implement the functions you have identified in your Triangles and in class discussion.

Extra Tasks

1. Write a function that counts how many times a specified character occurs in a string.

```
occ :: Char -> String -> Int
occ x = foldr (\c n -> if c == x then n + 1 else n) 0
```

2. A positive integer is *perfect* if it equals the sum of its factors, excluding the number itself. Write a function that returns the list of all perfect numbers up to a specified value.

```
isPerfect :: Int -> Bool
isPerfect n = sum [x|x<-[1..n-1],mod n x==0] == n

perfectTo :: Int -> [Int]
perfectTo n = [x|x<-[1..n], isPerfect x]
```

Homework

- Read the Essential Reading on Selection in Haskell in the next chapter.
- Do the Self-Test questions below

Self-Test: bring your answers to the next lesson You can use the Haskell compiler to check your answers.

1. Write a Haskell function that returns the **absolute value** of an integer; for example, abs(4) = 4 and abs (-4) = 4.

   a. Use an *if.. then.. else..*

```
abs1 :: Int -> Int
abs1 x = if x > 0 then x else -x
```

   b. Use *guards*

```
abs2 :: Int -> Int
abs2 x
    | x > 0 = x
    | otherwise = -x
```

2. Here is an incomplete definition for a *tail* function that returns the tail of a list (a list containing all but the first item).

   tail::[a]->[a]

   tail [a] = [a]

   tail (_:xs)=xs

   This definition uses pattern matching. The last line covers the general case (it means "the tail of a list is everything apart from the first item"). Think about when this definition will fail. Then complete the definition in the second line to cover this special case.

Caesar Cipher

Objectives

- Analyse a problem and identify how it can be broken down into a set of functions
- Develop and implement the required functions
- Homework: read about and practice using recursion in Haskell

The Problem

You need no introduction to this problem!

Essential Reading – Selection in Haskell

You will remember that all programming languages must support **selection**: the ability to follow alternative paths through an algorithm. How does Haskell achieve this? There are several choices.

If.. then .. else This is essentially the same as in imperative languages like Pascal:

```
doubleorhalve :: Float->Float
doubleorhalve x = if x<10 then 2*x else x/2
```

"Guards" If there are more than two alternatives then using "guards" is more readable:

```
doubleorhalve :: Float->Float
doubleorhalve x
| x==0 = x
| x<10 = 2*x
| otherwise = x/2
```

Guards are similar to case statements in Pascal but unlike case statements the alternatives do not have to be mutually exclusive: the first guard that evaluates to True will be executed and the others will be ignored.

Pattern Matching This third method involves making multiple definitions, covering different potential inputs:

```
inttostring :: Int->String
inttostring 1 ="One"
inttostring 2 ="Two"
inttostring 3 ="Three"
inttostring 4 ="Four"
inttostring x = "Not between one and four"
```

In the example above, there are five definitions for the function inttostring: four covering individual inputs and one as a "catch-all". The first one to match the given input will be executed: the catch-all must be defined last.

Task – the Caesar Cipher

Breaking the problem down into functions In your triangles, discuss and outline the functions you will need to encrypt a string using a Caesar Cipher:

- encryptChar
- encryptString

Now write code to implement the functions you have identified in your Triangles and in class discussion.

Hints

There are some library functions that will come in very useful:

ord :: Char -> Int returns the ASCII code of a character

chr :: Int -> Char returns the character corresponding to an ASCII code

toUpper :: Char -> Char converts a character to upper case

```
import Data.Char

encryptChar :: Int -> Char -> Char
encryptChar n x
    | ord x > 122 || ord x < 97 = x
    | ord x + n > 122 = chr $ ord x + n - 26
    | otherwise = chr $ ord x + n

encryptString :: Int -> String -> String
encryptString n = map (encryptChar n)
```

These have similar/identical names to their Pascal equivalents, so you shouldn't have much trouble understanding them. However, they are not in the standard library: you have to load the Data.Char library (called a **module** in Haskell), using the **import** command:

```
import Data.Char
```

Make the above the first line in your file.

Homework

- Read the "Essential Reading" on Recursion in the next chapter.
- Do the Self-Test exercises below.

Self-Test: bring your answers to the next lesson You can use the Haskell compiler to check your answers.

1. Write a Haskell function to return the nth Fibonacci number.

```
fibbonacci n
    | n < 2 = 1
    | otherwise = fibbonacci (n - 1) + fibbonacci (n - 2)
```

2. Write a Haskell function to **sum** the items in a numeric list.

```haskell
sum1 :: Num a => [a] -> a
sum1 = foldr (+) 0
```

3. Here is an incomplete definition of a function to insert an item into the correct place in a sorted list:

   insert :: Ord a -> [a]->[a]

   insert x [] = [x]

   insert x(y:ys)

   | x<=y = x:y:ys

   | otherwise = y:insert x ys

   Fill in the blanks.

   Easy: The first blank defines the result of inserting an item x into an empty list.

   Harder: The second blank is the recursive call. If x is greater than the first item in the list where does it get inserted?

Palindrome

Objectives

- Analyse a problem and identify how it can be broken down into a set of functions
- Develop and implement the required functions, which include the use of recursion
- Homework: read about and practice using higher order functions in Haskell

Another familiar one! Is a string a palindrome?

```haskell
palindrome :: Eq a => [a] -> Bool
palindrome xs = xs == reverse xs
```

Essential Reading – Recursion

You will remember that all programming languages must support the ability to repeat actions. In imperative languages this is most often achieved through iteration (looping) but you have recently learned about an alternative: recursion. In functional languages like Haskell recursion is the usual method of achieving repetition. Remember that iteration and recursion are equally capable: any repetitive action can be performed by either.

Here is the classic example of the factorial function written recursively:

```
factorial :: Int-> Int

factorial 0 = 1

factorial n = factorial (n-1) *n
```

This uses pattern matching to deal with the base case (n=0), which is a common technique in recursive Haskell functions, but you could use an *if.. then.. else* instead.

Another example, this time a function that processes a list:

```
product :: Num a => [a]-> a

product [] = 1

product (n:ns) = n * product ns
```

This is one possible definition of the library function *product,* which returns the product of the items in a numeric list.

The base case is straightforward: the product of an empty list is 1 (why not 0?).

However the general, recursive case needs some explanation:

The term *(n:ns)* describes the list as being made up (constructed) of a value, *n,* followed by some more values, *ns.* The colon *(:)* is the construction operator. (You met this in the first lesson.) It then recursively defines the product as being the first item, *n,* multiplied by the product of the remaining items, which does make sense, I hope.

Here is another example, this time to return the length of a list:

```
length :: [a]->Int

length []=0

length(_:xs)= 1 + length xs
```

In this example, the _ character is a wild card, standing for any value. So the recursive definition means "the length of a list made up of any value followed by some more values (the tail) is 1 plus the length of the tail". The wild card is used because the first value in the list is not used on the right-hand side of the definition (compare this with the definition of *product.*)

Task – Palindrome function

Breaking the problem down into functions

In your triangles, discuss and outline of the functions you will need to determine whether a string is a palindrome, using the **reverse and compare method**. Your method should ignore spaces in the string so that "bob" and "b o b" **are** counted as palindromes. You are **not** allowed to use the Prelude.Reverse function; write your own!

```haskell
stripSpaces :: String -> String
stripSpaces = filter (/= ' ')

revStr :: String -> String -> String
revStr s x
    | x == "" = s
    | otherwise = revStr (head x:s) $ tail x

palindrome :: String -> Bool
palindrome xs =
    let stripped = stripSpaces xs
    in stripped == revStr "" stripped
```

Now write code to implement the functions you have identified in your Triangles and in class discussion.

Homework

- Read the "Essential Reading" on Higher Order Functions in the next chapter.
- Do the Self-Test exercises below.

Self-Test: bring your answers to the next lesson You can use the Haskell compiler to check your answers.

1. The standard library function *isDigit* (in Data.Char) determines whether or not a character is a digit.

    a. Write a function which inputs a list of characters and outputs whether or not each one is a digit.

    b. Write a function which inputs a list of characters and outputs a list containing only those which are digits.

2. A function *sum* to sum a list of numbers can be defined like this:

    sum=foldr(+)0

    Write a definition in a similar style, using a fold, to define a function *product*, which returns the product of the items in a list of numbers.

```haskell
product :: Num a => [a] -> a
product = foldr (*) 1
```

3. The function *sumsquares* defined below sums the squares of all the numbers in a list of integers:

    sumsquares :: [Int]->Int

    sumsquares ns = sum(map(^2)ns)

a. Redefine the function so that it only operates on the odd numbers in the list (Use the library function *odd).*

```
sumOddSquares :: [Int] -> Int
sumOddSquares = sum . map(\x -> if odd x then x^2 else 0)
```

b. If you've not already done so, modify your solution to a., using the composition operator (.) to simplify your definition.

Base Converter Plus

Objectives

- Analyse a problem and identify how it can be broken down into a set of functions
- Develop and implement the required functions, which include higher-order functions

The Problem

Another one you are familiar with: convert a decimal integer to binary. Extend this to convert a string into a list of binary ASCII codes.

Example:

"Sheep" -> [[1,0,1,0,0,1,1],[1,1,0,1,0,0,0],[1,1,0,0,1,0,1],[1,1,0,0,1,0,1],[1,1,1,0,0,0,0]]

Essential Reading – Higher-Order Functions

A key feature of functional programming languages is that they treat functions as **"first-class objects"**. This means that functions are treated in exactly the same way as data. In fact, **a function is a data type**, which means that functions can be passed into other functions as inputs (parameters) and they can be passed out of functions as outputs:

A **higher-order function** is a function that can take another function either as an input or an output, or both, so function x in the diagram above is a higher-order function.

Let's look at some examples:

map Suppose we want to apply a function to every item in a list. A simple example would be to reverse every string in an list. You've already written a function to reverse a single string (and there's one in the standard library). We could of course modify our function or write a new one to process a whole list of strings but it would be easier if we didn't have to. The good news is we don't! There is a standard library function in Haskell called *map* which works like this:

For example:

>map reverse [“cat”, “dog”, “pig”]

returns [”tac”,”god”,”gip”]

map can be applied to any function/list pair as long as they are type-compatible.

map is clearly a higher-order function as it takes a function as an input.

filter Suppose we have a list of numbers from which we want to extract only the odd ones. There is a standard Boolean function called *odd* that returns True if its input is odd, False if it is even. We want to apply this to every item in our list, generating a new list of just the odd ones. We can use *filter* to do this:

In our example:

>filter odd [1..10]

returns [1,3,5,7,9]

filter can be applied to any Boolean function/list pair as long as they are type-compatible.

filter is clearly a higher-order function as it takes a function as an input.

foldr and foldl These are a bit trickier to understand. They are used to reduce a list to a single value by repeatedly applying the same operation. They “fold” the list up, item by item until only a single value is left as the output.

For example, the *sum* function can be written using a fold:

You can think of summing a list of numbers as initialising a running total, 0, and repeatedly applying addition to each item in the list, adding it to the running total:

sum [1,2,3,4,5] = 0+[1,2,3,4,5]

= 0+1+[2,3,4,5]

= 1+[2,3,4,5]

= 1+2+[3,4,5]

= 3+[3,4,5]

= 3+3+[4,5]

= 6+[4,5]

= 6+4+[5]

= 10+[5]

= 10+5

= 15

Of course, the repetition is achieved recursively. This could be defined like this:

sum[]=0

sum(x:xs) = x + sum xs

However, the *foldr* function provides a generic way of doing this. *foldr* takes a function, a list and a starting value as its inputs and repeatedly applies the function to each item in the list, keeping the running total as it goes and outputting the final value of the running total:

For example, the sum s=function can be defined like this:

sum = foldr (+) 0

This applies the addition function, +, to every item in the list in turn, adding it to the total, which is initialised to 0.

Foldr(ight) and *foldl(eft)* differ in the order they traverse the list and apply the operator. Sometimes either can be used but sometimes only one will work correctly, depending on the input function.

Like *map* and *filter*, folds are useful generic functions which can be used to apply existing functions to a range of problems without having to start from scratch or make major modifications.

Like *map* and *filter*, folds are clearly higher-order functions as they take a function as an input.

Functions as outputs: "curried" functions We've seen three examples of higher-order functions that take functions as inputs. What about functions that generate functions as their output?

Surprisingly, many Haskell functions do this; in fact, *any function which appears to take more than one input!* Strictly speaking all Haskell functions take only one input. This sounds ridiculous. Even a function as simple as addition has to have two inputs: you can't add a number to nothing (you can add it to zero but that's not the same). 3+ is meaningless, or is it? How about thinking of 3+ as being a *function*? You could describe it as "the function which adds 3 to something" where "something" is its input:

Taking this idea further, we can break down the addition function into two steps:

Step 1: the addition function takes an input, 3 in our example and outputs the *3+* function:

Step 2: the freshly-minted 3+ function takes an input, let's say 4 for example, and completes the addition to give an output, 7.

Sounds crazy? This turns out to be a useful concept. Functions that take one input at a time, returning intermediate functions as results, are called "curried functions". (Haskell Curry was the mathematician after whom Haskell is named. He developed these ideas.)

Curried functions are more flexible than functions on multiple inputs, because useful functions can often be made by *partially applying* a curried function. For example the *take* function that selects members from a list is curried. First it takes an integer as an input, returning a function:

This *partially applied function, take 2,* can then be used as the input to a different function, for example, *map*:

For example: >map (take 2) [[1,2,3],[4,6,8],[5,7,9,11]]

returns [[1,2],[4,6],[5,7]]

Function Composition You are familiar with the idea of one function calling another. For example, the expression *sqrt(cos(x))* means that the *cos* function is called first with an input of x and the output of that function is used as the input of the *sqrt* function:

Effectively the two functions are being combined to make a single function. This process is called *function composition*. Composition involves taking two (or more) functions as inputs and producing another function as an output, so you can think of this as a higher-order function:

For simplicity, *compose* has been shown as a single-step function with two inputs, though of course it could (and should, to make clear the order in which the functions are called) be expressed as a two-step curried function.

Haskell has a standard function to compose two functions. It's called by using the operator (in practice just a full stop). To define a new function called *myfunc*, by composing *cos* and *sqrt,* we just write this:

```
myfunc= sqrt .cos
```

Note that the inputs are just the names of the functions you want to compose; you don't need to include the inputs to those functions. But do make sure you get the order correct. The function will be composed in right-to-left sequence as in the usual mathematical notation.

Using composition often results in simpler, cleaner syntax. For example, the *myfunc* function above could also be written like this, with brackets and inputs:

```
myfunc x = sqrt(cos x)
```

The Problem – Convert a string to Binary Character Codes

Breaking the problem down into functions

In your triangles, discuss and outline the functions you will need to convert a string into a list of binary character codes.

Now write code to implement the functions you have identified in your Triangles and in class discussion.

```
import Numeric
import Data.Char

toBinary :: Int -> String
toBinary x = showIntAtBase 2 ("01"!!) x ""

charToBinary :: Char -> String
charToBinary = toBinary . ord

stringToBinaryValues :: String -> [String]
stringToBinaryValues = map charToBinary
```

Extra Task

Return to your Caesar Cipher and Palindrome scripts and simplify them by using filters to remove the spaces from the input text.

Big Data

Objectives

- Understand the meaning of the term "Big Data" and its characteristics
- Understand why functional programming is well-suited to processing Big Data
- Be familiar with fact-based models and graph schemas for representing Big Data
- Gain some practical experience of MapReduce and Machine Learning techniques for processing Big Data

Homework – Introduction to Big Data

- Read Chapter 11.1, Sections 1 and 2, pp. 543-555 in Bond
- Answer these questions:

1. Name and briefly explain the three defining characteristics of Big Data.
2. What is meant by a distributed file system?
3. What is meant by fault-tolerant?

Lesson Tasks – MapReduce and Functional Programming

Discuss in your triangles :

1. Explain why parallel processing is appropriate for implementing the MapReduce technique.
2. What features of functional programming make it particularly suitable for writing software to process Big Data?

Using MapReduce with MATLAB

Although not a purely functional language (it is very much a *multi-paradigm* language) MATLAB has features which make it suitable for handling large datasets, including powerful matrix manipulations, MapReduce functions and machine learning algorithms.

Inspecting the data

- Open the file, *airlinesmall.csv* in Excel. How many rows of data are there?
- Now, run MATLAB, open the same file and try to import the data:

  What does the error dialog say?
- **Cancel the Import**. MATLAB lets you set up a *datastore* to deal with large datasets like this. This is essentially a pointer or reference to the data, enabling you to work with it by importing it in chunks, rather than all at once. Set up a datastore for the airline data like this:

  »airlineds=datastore('airlinesmall.csv', 'TreatAsMissing', 'NA');

  You can preview the datastore like this:

  »preview(airlineds)

  This shows you the first few rows, but still without loading all the data into memory.

  Let's just focus on one column, the delay for each flight:

  »airlineds.SelectedVariableNames = 'ArrDelay';

  »preview (airlineds)

Applying a MapReduce technique to the data For this exercise you are going to apply the MapReduce technique to find the maximum arrival delay in the data.

The MapReduce method consists of two phases:

- Map: break the data up into chunks and find the maximum for each chunk. Store the maxima temporarily.

- Reduce: merge the maxima for each chunk to find the overall maximum.

http://uk.mathworks.com/help/matlab/import_export/mrworkflow.png

Extra details

- In MATLAB the chunk size is a property of the datastore; the default is 20000 rows.
- The map function is called many times, once for each chunk, and in our example, stores the maximum value for that chunk.
- In our example, because there is only one value stored for each chunk (the maximum delay for that chunk), the reduce function will only be called once.

You are provided with the map and reduce functions, called *maxArrivalDelayMapper* and *maxArrivalDelayReducer.* Call them up like this:

» maxDelay = mapreduce(airlineds, @maxArrivalDelayMapper, @maxArrivalDelayReducer);

This command applies the map and reduce functions to the datastore and stores the result in *maxDelay* (also a datastore, allowing for the possibility that the results could also be too large to fit in memory).

You can examine the results like this:

» preview (maxDelay)

What is the maximum delay time?

Extra

To calculate the **mean** delay, what values would the map function need to store?

Check your answer by looking at the supplied functions: *meanArrivalDelayMapper* and *meanArrivalDelayReducer.* Call them up using the *mapreduce* function. What is the mean delay?

Homework – Fact-based models and Graph Schemas

- Read Chapter 11.1, Section 3, pp. 556-560 in Bond
- Answer these questions:

1. Describe the fact-based model.
2. In the fact-based model, what is represented by:
    a. A node
    b. An edge

c. A property

Lesson Tasks – Fact-based models and Graph Schemas

Discuss in your triangles :

1. Why does the fact-based model require Big Data techniques?
2. What are the advantages of the graph-based model over the relational model?

Machine Learning and Modelling

Machine Learning techniques are often used in Big Data scenarios. If the **volume** and **velocity** of incoming data are large then it is often impossible to analyse it manually and it becomes essential to develop an algorithm to do the analysis (see pages 552-554 in Bond).

The key steps in Machine Learning are:

- Capture some sample data
- "Train" the algorithm by using some of the sample data to develop a model
- Test the model using the remaining sample data

If the test is successful then the model can be used with new data. Using the model to make predictions will be much faster than manual techniques.

Machine Learning: Concrete Strength Example The spreadsheet, Concrete_Data.xls, contains data on the strength of various mixes of concrete. Each mix contains seven different ingredients and the age of the mix provides an eighth variable. You are going to use both manual and machine learning techniques to investigate the relationship between the variables and the strength of the concrete and develop a model.

Manual analysis Import data from file Concrete_Data.xls

a) As nine separate column vectors

b) As a numeric matrix, containing all the data

After the import, you should have the following variables in your workspace:

Now, try to find a correlation between the ingredients and the resulting strength of the mix:

a) **Visual inspection of plots**

Produce the following plots:

1. Concrete strength against concrete age

2. Concrete strength against the quantity of cement
3. Concrete strength against the quantity of superplasticizer

Example: This command will generate a scatter plot of strength against age:

scatter(Age, Strength)

Can you deduce what concrete strength most depends on?

Even if you produce plots of concrete strength against all other variables, it will still be hard to visually compare them and deduce the dependencies. Instead, a better way of understanding the dependencies between the variables is by looking at correlations[1]. Without going into too many details about formal statistical definition of correlation, we will accept correlation as a measure of (linear) dependency between the variables.

b) **Using correlation coefficients**

Using MATLAB function corrcoef, calculate pairwise correlations between the concrete variables and store them in a matrix of the form:

$$\begin{matrix} c_{11} & c_{12} & c_{13} \cdots \\ c_{21} & c_{22} & c_{23} \cdots \\ c_{31} & c_{32} & c_{33} \cdots \end{matrix}$$

Where c_{ij} is the correlation between the i^{th} and the j^{th} concrete variable:

» corrmat = corrcoef(ConcreteData)

The matrix, corrmat, contains the correlation coefficients for every pair of variables (+1 is the maximum possible positive correlation, -1 is the maximum possible negative correlation, 0 means there is no correlation).

Although it is now much easier to compare the dependencies between the different pairs of variables, the dependencies may still not be obvious. To visualise the correlation matrix, use the provided custom-coded function plotCorr:

» labels = { 'Cement', 'BlastFurnaceSlag', 'FlyAsh' , 'Water', 'Superplasticizer', 'CoarseAggregate', 'FineAggregate', 'Age', 'Strength'} % make this in one line

» short_labels = plotCorr(corrmat,labels);

The chart uses colour coding to represent the correlation coefficients. Can you now tell what concrete strength most depends on?

[1]Correlation between two variables is obtained by dividing the covariance of the two variables by the product of their standard deviations.

Using Machine Learning techniques You are now going to use the data to develop a model based on the data. This model can then be used to predict the strength of any given mix of concrete.

The steps are:

- Use some of the data to generate a model
- Use the remaining data to test the model

If the test is successful then the model can be used with new data. Using the model to make predictions will be much faster than manual techniques.

First, split the existing data into training and testing sets for prediction. The training set should contain 70% of available data, and the testing set should contain the remaining 30% of the data. Bear in mind that you will need to define both inputs and outputs for training and testing.

Let' use the first 70% of rows for training and the last 30% of rows for testing:

» XTrain = ConcreteData(1:721, 1:8); %the inputs: ingredients

» YTrain = ConcreteData(1:721, end); %the output: concrete strength

» XTest = ConcreteData(722:end, 1:8);

» YTest = ConcreteData(722:end, end);

MATLAB includes a lot of different modelling algorithms. For this exercise you are going to use a Linear Regression technique.

First, create the model:

» modelLR = fitlm(XTrain, YTrain);

You can have a look at it if you like:

» disp(modelLR)

(You don't need to understand this!)

Now test the model, to see if its predictions for the strength of the concrete mix match the measurements in the test data:

» yFitLR = predict(modelLR,XTest);

» figure

» scatter(YTest, yFitLR, 'filled')

» hold on

» plot([0 max(YTest)] , [0 max(YFitLR)])

» hold off

The chart shows the model's predictions on the x-axis and the measured strengths in the test data on the y-axis.

How good is the model? You can get a statistical measure for this:

corr(YTest,yFitLR)

This returns the correlation between the model's predictions and the measured strengths. A value of 1 would mean the model was perfect, a value of 0 would mean that its predictions are no better than random.

What is the correlation?

You now have a model that could be used to predict very quickly the strength of any concrete mix.