Data Handling: Import, Cleaning and Visualisation

Lecture 2: A Brief Introduction to Data and Data Processing

Prof. Dr. Ulrich Matter

27/09/2018

1 Data in human history

In order to better understand the role of data in today's economy and society, we have a look at the usage forms and purposes of data records in human history. In a second step, we look at how a computer processes data.

Throughout human history, the recording and storage of data has primarily been motivated by measuring, quantifying, and keeping record of both our social and natural environments. Early on, the recording of data has been related to economic activity and scientific endeavor. The neolithic transition from hunter-gatherer societies to agriculture and settlements (the economic development sometimes referred to as the 'first industrial revolution') came along with a division of labor and more complex organizational structures of society. The change to agricultural food production had on the one hand the effect that more mouths could be fed, but on the other hand also implied that food production would need to follow a more careful planning (e.g., the right time to seed and harvest) and that the produced food (e.g., grains) would partly be stored and not consumed entirely on the spot. It is believed that partly due to these two practical problems, keeping track of time and keeping record of produced quantities, neolithic societies started to use signs (numbers/letters) carved in stone or wood (Hogben 1983). Keeping record of the time and later measuring and keeping record of produce quantities in order to store and trade likely led to the first 'data sets'. Simultaneously, the development of mathematics, particularly geometry took shape.

2 Processing data: simple calculations in numeral systems

In order to keep track of calculations with large numbers, humans started to use mechanical aids such as pebbles or shells and later developed the counting frame ('abacus').[1] We can understand the counting frame as a simple mechanical tool to process numbers (data). In order to use the abacus properly, we have to agree on a standard regarding what each column of the frame represents, as well as how many beads each column can contain. In other words, we have to define what the *base* (or 'radix') of the frame's numeral system is. For example, the Roman numeral system is essentially of base 10, with 'I' representing one, 'X' representing ten, 'C' representing one hundred, and 'M' representing one thousand. @**??**fig:abacus) illustrates how a counting frame based on this numeral system works (examples are written out in Arabic numbers). The first column on the right represents units of $10^0 = 1$, the second $10^1 = 10$, and so forth.

From inspecting @**??**fig:abacus) we recognize that the columns of the abacus are the positions of the digits, which also signify the power of 10 with which the digit is multiplied: $139 = (1 \times 10^2) + (3 \times 10^1) + (9 \times 10^0)$.[2] In addition, a base of 10 means that the system is based on 10 different signs (0, 1, ..., 9) with which we distuingish one-digit numbers. Further, we recognize that each column in the abacus has 10 beads.

The numeral system with base 10 (the 'decimal system') is what we use to measure/count/quantify things in our everyday life. Moreover, it is what we normally work with in applied math and statistics (at least in the realm of undergraduate economics). However, the simple but very useful concept of the counting frame also works well for numeral systems with other bases. Historically, numeral systems with other bases existed in

[1] See Hogben (1983), Chapter 1 for a detailed account of the abacus' origin and usage.

[2] Starting with position/power 0.

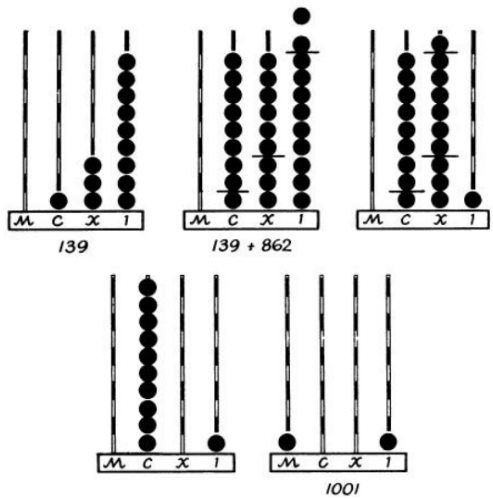

Figure 1: A simple abacus. Source: Hogben (1983).

various cultures. For example, ancient cultures in Mesopotamia used different forms of a sexagesimal system (base 60), consisting of 60 different signs to distinguish 'one-digit' numbers.

Note that this logic holds both ways: if a numeral system only consists of two different signs, it follows that the system has to be of base 2 (i.e., a 'binary system'). As it turns out, this is exactly the kind of numeral system that gets relevant once we use electronic tools, that is, digital computers, to calculate and process data.

2.1 The binary system

Anything related to electronically processing (and storing) *digital data*, has to be built on a binary system. The reason is that a microprocessor (similar to a light switch) can only represent two signs (states): *on and off*. We usually refer to 'off' with the sign '0' and to 'on' with the sign '1'. A numeral system consisting only of the signs 0 and 1 thus must be of base 2. It follows that an abacus of this binary system has columns $2^0 = 1$, $2^1 = 2$, $2^3 = 4$, and so forth. Moreover, each column has only one bead (1) or none (0). Nevertheless we can express all (natural) numbers from the decimal system.[3] For example, the number 139 which we have expressed with the 'decimal abacus', would be expressed as follows with a base 2 system:

$$(1 \times 2^7) + (1 \times 2^3) + (1 \times 2^1) + (1 \times 2^0) = 139.$$

More precisely, when including all columns of our binary system abacus (that is, including also all the columns set to 0), we would get the following:

$$(1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 139.$$

Now, compare this with the abacus in the decimal system above. There, we've set the third column to 1, the second column to 3 and the first column (from the right) to 9: 139. If we do the same in the binary system (where we can set each column either to 0 or 1) we get 10001011. That is, the number 139 in the decimal system corresponds to 10001011 in the binary system.

[3]Representing fractions is much harder in a binary system than representing natural numbers. The reason is that fractions such as $1/3 = 1.333..$ actually constitute an infinite sequence of 0s and 1s. The representation of such numbers in a computer (via so-called 'floating point' numbers) is thus in reality not a 100% accurate.

How can a computer know that? The only way a computer correctly prints the three symbols `139` to our computer screen when dealing with the binary expression `10001011`, is that a mapping between binary coded values and decimal numbers is 'hard-coded' into the computer. Since a computer can only understand binary expressions (1s and 0s), we have to define a *standard* of how `0`s and `1`s correspond to symbols, colors, etc. that we see on the screen.

Of course, this does not change the fact that any digital data processing is in the end happing in the binary system (as computers can only process 0s and 1s). But, in order to avoid having to work with a keyboard consisting only of an on/off (1/0) switch, low-level standards that define how symbols like `3`, `A`, `#`, etc. correspond to expressions in 0s and 1s help us to interact with the computer. It makes it easier to enter data and commands into the computer as well as understand the output (i.e., the result of a calculation performed on the computer) on the screen. The standard defining how our number symbols in the decimal system correspond to binary numbers (again, reflecting the idea of an abacus) can be illustrated in the following table:

Number	128	64	32	16	8	4	2	1
0 =	0	0	0	0	0	0	0	0
1 =	0	0	0	0	0	0	0	1
2 =	0	0	0	0	0	0	1	0
3 =	0	0	0	0	0	0	1	1
...								
139 =	1	0	0	0	1	0	1	1

2.2 The hexadecimal system

From the above table we also recognize that binary numbers can become quite long rather quickly (have many digits). In Computer Science it is, therefore, quite common to use another numeral system to refer to binary numbers: the *hexadecimal* system. In this system we have 16 symbols available, consisting of `0-9` (used like in the decimal system) and `A-F` (for the numbers 10 to 15). Because we have 16 symbols available, each digit represents an increasing power of 16 (16^0, 16^1, etc.). The decimal number 139 is expressed in the hexadecimal system as follows

$$(8 \times 16^1) + (11 \times 16^0) = 139.$$

More precisely following the convention of the hexadecimal system used in Computer Science (where `B` stands for `11`), it is:

$$(8 \times 16^1) + (B \times 16^0) = 8B = 139.$$

Hence, `10001011` in the binary system is `8B` in the hexadecimal system and `139` in the decimal system. The primary use of hexadecimal notation when dealing with binary numbers is the more 'human-friendly' representation of binary coded values. First, it is shorter than the raw binary representation (as well as the decimal representation). Second, with a little bit of practice it is much easier for humans to translate forth and back between hexadecimal and binary notation than it is between decimal and binary. This is because each hexadecimal digit can directly be converted into its four-digit binary equivalent (from looking at the table above): $8 = 1000$, $B = 11 = 1011$, thus `8B` in hexadecimal notation corresponds to `10001011` (`1000 1011`) in binary coded values.

3 Character encoding

Obviously, computers are not only used to process numbers but in a wide array of applications they are used to process text. Most fundamentally, when writing computer code, we type in commands in the form of text

consisting of common alphabetical letters. How can a computer understand text if it only understands `0`s and `1`s? Well, again we have to agree on a certain standard of how `0`s and `1`s correspond to characters/letters. While the conversions of integers between different numeral systems follow all essentially the same principle (~ the idea of a 'digital' abacus), the introduction of standards defining how `0`s and `1`s correspond to specific letters of different human languages is way more arbitrary.

Today, there are many standards defining how computers translate `0`s and `1`s into more meaningful symbols like numbers, letters, and special characters in various natural (human) languages. These standards are called *character encodings*. They consist of what is called a 'coded character set', essentially a mapping of unique numbers (in the end in binary coded values) to each character in the set. For example, the classical ASCII (American Standard Code for Information Interchange) assigns the following numbers to the respective characters:

Binary	Hexadecimal	Decimal	Character
0011 1111	3F	63	`?`
0100 0001	41	65	`A`
0110 0010	62	98	`b`

Note that the convention that `0100 0001` corresponds to `A` is simply true by definition of the ASCII character encoding.[4] It doesn't follow from any law of nature or fundamental logic. But having such standards (and widely accept them) is paramount to have functioning software and meaningful data sets that can be used and shared across many users and computers. If we write a text and store it in a file based on the ASCII standard and that text (which, under the hood, only consists of `0`s and `1`s) is read on another computer only capable of mapping the binary code to another character set, the output would likely be complete gibberish.

In practice, the software we use to write emails, browse the Web, or conduct statistical analyses all have some of these standards built into them and usually we do not have to worry about encoding issues when simply interacting with a computer through such programs. The practical relevance of such standards becomes much more relevant once we *store* or *read* previously stored data/computer code.

4 Computer code and text files

Understanding the fundamentals of what digital data is and how computers process them is the basis for approaching two core themes of this course: (I) How *data* can be *stored* digitally and be read by/imported to a computer (this will be the main subject of the next lecture), and (II) how we can give instructions to a computer by writing *computer code* (this will be the main topic of the first two exercises/workshops).

In both of these domains we mainly work with one simple type of document: *text files*. Text files are simply a collection of characters (with a given character encoding). Following the logic of how binary code is translated forth and back into text, they are a straightforward representation of the underlying information (`0`s and `1`s). They are used to store both structured data (e.g., tables), unstructured data (e.g., plain texts), or semi-structured data (such as websites). Similarly, they are used to write and store a collections of instructions to the computer (i.e., computer code) in any computer language.

From our every-day use of computers (notebooks, smartphones, etc.) we are accustomed to certain software packages to write text. Most prominantly, we use programs like Microsoft Word or email clients (Outlook, Thunderbird, etc.). However, these programs are a poor choice for writing/editing plain text. Such programs tend to add all kinds of formatting to the text and use specific file formats to store formatting information in addition to the raw text. In other words, when using this type of software, we actually do not only write plain text. Any program to read data or execute computer code, however, expects only plain text. Therefore, we must only use *text editors* to write and edit text files. In this course we will use either Atom or RStudio.

[4] Each character digit is expressed in 1 byte (8 `0`/`1` digits). The ASCII character set thus consists of $2^8 = 256$ different characters.

5 Data processing basics

Murrell (2009) chapter 9, pages 199-204.

References

Hogben, Lancelot. 1983. *Mathematics for the Million*. New York: W.W Norton & Company.

Murrell, Paul. 2009. *Introduction to Data Technologies*. London, UK: CRC Press.