Text encoding (theory)

Hauke Licht
University of Cologne

April 17, 2024

Getting the basics straight

- **Characters** are abstractions without inherent visual representation; they are the concepts like ‘A’, ‘new line’.
- **Computers** store and manipulate text using a sequence of numbers; encoding converts characters into these numbers.
- The representation of characters in computer systems are encoded as **bytes**.
- A **byte** is composed of typically 8 bits.
- A **bit** is a binary digit, which can be either 0 or 1.
- Hence, one byte can create 256 (2^8) different values

⇒ *Encodings* **map characters to bytes**.

Getting the basics straight

⇒ ***Encodings*** **map characters to bytes.**

Example

- the `ASCII` encoding covers (mainly) English alphabets and common symbols
- the character `A` has the 65th position in ASCII's table and is encoded with the byte `01000001` (65 in binary)

Je suis désolé, je n’ai pas compris?

- ASCII doesn’t include characters from languages such as French, German, Greek, which use accented characters (e.g. “à”), different alphabets (e.g., “ä”, “ß”, etc.), or even characters from different scripts (e.g., “α”, “и”, etc.)
- Extended ASCII attempts to include more characters by using extra bits, but it leads to inconsistencies across different systems.
- Various encoding systems were developed like ISO-8859 and Windows-1252 to cater to European alphabets, each differing in the characters they include or exclude.

Multi-byte encodings

- Single-byte encdigns can at most represent 256 (2^8) characters.
- To include more characters than single bytes allow, multi-byte encoding systems were introduced.
- In a multi-byte system, the number of bytes to represent a character can vary from one to several.
- Shift JIS (Japanese Industrial Standards) and GB2312 for Chinese are examples of multi-byte encoding, using different byte lengths for different characters, which complicates text processing.

Unicode to the ~~rescue~~ confusion

- **Unicode** aims to uniquely represent every character from every language in a consistent manner.
- It assigns a unique number, called **code point**, to each character and symbol across languages and scripts (see https://unicode.org/charts).
- Unicode can be encoded into bytes using methods like **UTF-8**, UTF-16, which vary in the number of bytes used per character depending on the specific characters.
- You can view the Unicode table at https://unicode-table.com/en/

UTF-8 encoding

- **UTF-8** is a variable-width encoding that uses 1 to 4 bytes (i.e. between 8 and 4×8 bits) to represent characters.
- It is backward-compatible with ASCII, as the first 128 characters (0-127) are the same as ASCII.
- Characters beyond ASCII are represented using multiple bytes, with the most common characters using fewer bytes.
- you can view how UTF-8 encodes Unicode characters at https://www.fileformat.info/info/charset/UTF-8/list.htm

Best practice

When collecting, sharing and collaborating on text datasets, always use UTF-8 encoding

- this is the default on Unix-based systems (MacOS and Linux)
- on Windows, you need to change the system settings (see here)

UTF-8 encoding

Exercise

- Go to https://www.fileformat.info/info/charset/UTF-8/list.htm
- Find the code point for the character A
- Find the code points to write your full name
- Find out what Unicode code point is used to represent the 😜 emoji ("Face with Uneven Eyes and Wavy Mouth")

My solution

- the letter A's code point is `U+0041`
- my name (*Hauke Licht*) has the Unicode code points `U+0048` `U+0041` `U+0055` `U+004B` `U+0045` `U+0020` `U+004C` `U+0049` `U+0043` `U+0048` `U+0054` (note: `U+0020` is a white space)
- the woozy face emoji has code point `U+1F974` (see https://emojipedia.org/woozy-face#technical)

Problems

- Converting between encodings can result in data loss or corruption if not handled properly, which is a common issue in text processing.
- The misinterpretation of text files' encoding (e.g., treating a UTF-8 file as ASCII) leads to display errors and data corruption.

Best practices

1. Always **use UTF-8** encoding, always, forever, for everything (unless you have a good reason not to).
    1. Use file reading and writing functions in the `readr` R package (always uses UTF-8)
    2. When processing text data with other computer programs (e.g., Excel), make sure that you use UTF-8 for in- and export
2. When loading **external data** into R, always tabulate the unique characters in it the relevant text column to check if there are any obvious anomalies (see notebook "broken_characters.qmd")

Good to know (I)

Unicode code points can be represented in different ways:

- `U+0041` (hexadecimal)
- `U+41` (hexadecimal without leading zeros)
- `&#65;` (decimal, usefull for HTML encoding)
- `&#x41;` (hexadecimal)

Good to know (II)

Characters in the Unicode table are organized in different categories:

- **Letter** (e.g., A, α, и)
- **Number** (e.g., 1, IV, ¼, ①)
- **Punctuation** (e.g., !, ?, ;, …)
- **Symbol** (e.g., @, #, €, ©)
- **Separator** (e.g., space, tab, line break)

see https://www.compart.com/en/unicode/category

Note: knowing them becomes useful when working with regular expressions.

Summary

- **Characters** are abstract concepts, while **bytes** are the actual storage units for text in computers.
- **Encodings** map characters to bytes
- **Unicode** aims to represent all characters in all languages consistently.
- **UTF-8** is a variable-width encoding that uses 1 to 4 bytes to represent characters.
- **Best practice**: always use UTF-8 encoding when working with text data.

Examples and Exercises

Examples and Exercises

see notebook “broken_characters.html”